

Jc518 U.S. PTO  
09/243101  
02/02/99

# Java Card Virtual Machine Specification

---

*Java Card Version 2.1*

*Draft 2a*



Sun Microsystems, Inc.  
901 San Antonio Road  
Palo Alto, CA 94303 USA  
415 960-1300 fax 415 969-9131

Draft 2a, January 29, 1999

**Best Available Copy**

**This Page Blank (uspto)**

Copyright © 1998 Sun Microsystems, Inc.

901 San Antonio Road, Palo Alto, CA 94303 USA

All rights reserved. Copyright in this document is owned by Sun Microsystems, Inc.

Sun Microsystems, Inc. (SUN) hereby grants to you at no charge a nonexclusive, nontransferable, worldwide, limited license (without the right to sublicense) under SUN's intellectual property rights that are essential to practice the Java™ Card™ Virtual Machine 2.1 Specification ("Specification") to use the Specification for internal evaluation purposes only. Other than this limited license, you acquire no right, title, or interest in or to the Specification and you shall have no right to use the Specification for productive or commercial use.

#### RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-1(a).

SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.

#### TRADEMARKS

Sun, the Sun logo, Sun Microsystems, JavaSoft, JavaBeans, JDK, Java, Java Card, HotJava, HotJava Views, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, EmbeddedJava, PersonalJava, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, ToolTalk, Ultra, Ultracomputing, Ultraserver, Where The Network Is Going, Sun WorkShop, XView, Java WorkShop, the Java Coffee Cup logo, and Visual Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.



Please  
Recycle



**Best Available Copy**

**This Page Blank (uspto)**

# Contents

---

**Contents iii**

**Figures vii**

**Tables ix**

- 1. Introduction 1**
  - 1.1 Motivation 1
  - 1.2 The Java Card Virtual Machine 2
  - 1.3 Java Language Security 5
  - 1.4 Java Card Runtime Environment Security 6
- 2. A Subset of the Java Virtual Machine 7**
  - 2.1 Why a Subset is Needed 7
  - 2.2 Java Card Language Subset 7
    - 2.2.1 Unsupported Items 8
    - 2.2.2 Supported Items 10
    - 2.2.3 Optionally Supported Items 12
    - 2.2.4 Limitations of the Java Card Virtual Machine 12
  - 2.3 Java Card VM Subset 14
    - 2.3.1 class File Subset 14
    - 2.3.2 Bytecode Subset 16

2.3.3 Exceptions 18

**3. Structure of the Java Card Virtual Machine 23**

- 3.1 Data Types and Values 23
- 3.2 Words 24
- 3.3 Runtime Data Areas 24
- 3.4 Applet Contexts 24
- 3.5 Frames 25
- 3.6 Representation of Objects 25
- 3.7 Special Initialization Methods 25
- 3.8 Exceptions 26
- 3.9 Binary File Formats 26
- 3.10 Instruction Set Summary 26
  - 3.10.1 Types and the Java Card Virtual Machine 27

**4. Java Card Naming 31**

- 4.1 Overview of Token-based Linking 31
  - 4.1.1 Externally Visible Items 31
  - 4.1.2 Private Tokens 32
  - 4.1.3 The Export File and Conversion 32
  - 4.1.4 References – External and Internal 33
  - 4.1.5 Installation and Linking 33
- 4.2 Token Assignment 33
  - 4.2.1 Token Details 34

**5. The Export File Format 37**

- 5.1 Export File Name 38
- 5.2 Export File 38
- 5.3 Constant Pool 40
  - 5.3.1 CONSTANT\_Package 40
  - 5.3.2 CONSTANT\_Interfacesref 42

5.3.3	CONSTANT_Integer	43
5.3.4	CONSTANT_Utf8	43
5.4	Classes	44
5.5	Fields	47
5.6	Methods	49
5.7	Attributes	50
5.7.1	ConstantValue Attribute	51
<b>6.</b>	<b>The Cap File Format</b>	<b>53</b>
6.1	Component Model	54
6.1.1	Containment in a JAR File	55
6.1.2	Defining New Components	55
6.2	Header Component	56
6.3	Directory Component	59
6.4	Applet Component	61
6.5	Imports Component	62
6.6	Constant Pool Component	63
6.6.1	CONSTANT_Classref	64
6.6.2	CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, and CONSTANT_SuperMethodref	66
6.6.3	CONSTANT_StaticFieldref and CONSTANT_StaticMethodref	68
6.7	Class Component	70
6.7.1	interface_info and class_info	71
6.8	Method Component	76
6.8.1	exception_handler_info	77
6.8.2	method_info	78
6.9	Static Field Component	80
6.10	Reference Location Component	83
6.11	Export Component	85
6.12	Descriptor Component	87
6.12.1	class_descriptor_info	88

6.12.2	field_descriptor_info	89
6.12.3	method_descriptor_info	91
6.12.4	type_descriptor_info	93
<b>7.</b>	<b>Java Card Virtual Machine Instruction Set</b>	<b>95</b>
7.1	Assumptions: The Meaning of “Must”	95
7.2	Reserved Opcodes	95
7.3	Virtual Machine Errors	96
7.4	Security Exceptions	96
7.5	The Java Card Virtual Machine Instruction Set	97
<b>8.</b>	<b>Tables of Instructions</b>	<b>217</b>
	<b>Glossary</b>	<b>219</b>

# Figures

---

FIGURE 1-1	Java Card Applet Conversion and Distribution	3
FIGURE 1-2	Java Card Applet Distribution and Installation	4
FIGURE 6-1	Static Field Order Map	80
FIGURE 7-1	An example instruction page	97

*This Page Blank (uspto)*

# Tables

---

TABLE 2-1	Unsupported constant pool tags	14
TABLE 2-2	Supported constant pool tags.	15
TABLE 2-3	Support of checked exceptions	19
TABLE 2-4	Support of runtime exceptions	20
TABLE 2-5	Support of errors	20
TABLE 3-1	Type support in the Java Card Virtual Machine Instruction Set	27
TABLE 3-2	Storage types and computational types	29
TABLE 4-1	Token Range, Type and Scope	34
TABLE 5-1	Constant Pool Tags	40
TABLE 5-2	Package Flags	41
TABLE 5-3	Class access and modifier flags	45
TABLE 5-4	Field access and modifier flags	48
TABLE 5-5	Method access and modifier flags	50
TABLE 6-1	Component Tags	54
TABLE 6-2	JAR File Names	55
TABLE 6-3	Package Flags	57
TABLE 6-4	Constant Pool Tags	64
TABLE 6-5	Interface and Class Info Flags	71
TABLE 6-6	Method flags	79

TABLE 6-7	Static Field Sizes	81
TABLE 6-8	Array Types	82
TABLE 6-9	One-byte Reference Location Example	84
TABLE 6-10	Class Access and Modifier Flags	88
TABLE 6-11	Field Access and Modifier Flags	90
TABLE 6-12	Primitive Type Descriptor Values	91
TABLE 6-13	Method Access and Modifier Flags	92
TABLE 6-14	Type Descriptor Values	94
TABLE 8-1	Instructions by Opcode Value	217
TABLE 8-2	Instructions by Opcode Mnemonic	218

# Preface

---

Java™ Card™ technology combines a subset of the Java programming language with a runtime environment optimized for smart cards and similar small-memory embedded devices. The goal of Java Card technology is to bring many of the benefits of Java software programming to the resource-constrained world of devices like smart cards.

The Java Card *platform* is defined by three specifications: this *Java Card Virtual Machine Specification*, the *Java Card 2.1 Application Programming Interface*, and the *Java Card Runtime Environment (JCRE) 2.1 Specification*.

This specification describes the required behavior of the Java Card 2.1 Virtual Machine (VM) that developers should adhere to when creating an *implementation*. An implementation within the context of this document refers to a licensee's implementation of the Java Card Virtual Machine (VM), Application Programming Interface (API), Converter, or other component, based on the Java Card technology specifications. A Reference Implementation is an implementation produced by Sun Microsystems, Inc. Programs written for the Java Card platform are referred to as Java Card applets.

## Who Should Use This Specification?

This document is for licensees of the Java Card technology to assist them in creating an implementation, developing a specification to extend the Java Card technology specifications, or in creating an extension to the Java Card Runtime Environment (JCRE). This document is also intended for Java Card applet developers who want a greater understanding of the Java Card technology specifications.

---

## Before You Read This Specification

Before reading this document, you should be familiar with the Java programming language, the Java Card technology specifications, and smart card technology. A good resource for becoming familiar with Java technology and Java Card technology is the Sun Microsystems, Inc. website, located at: <http://java.sun.com>.

---

## How This Book Is Organized

**Chapter 1, "Introduction,"** provides an overview of the Java Card Virtual Machine architecture.

**Chapter 2, "A Subset of the Java Virtual Machine,"** describes the subset of the Java programming language and Virtual Machine that is supported by the Java Card specification.

**Chapter 3, "Structure of the Java Card Virtual Machine,"** describes the differences between the Java Virtual Machine and the Java Card Virtual Machine.

**Chapter 4, "Java Card Naming,"** describes the basic uses and requirements of link tokens, and provides details of how tokens are used during software production and installation.

**Chapter 5, "The Export File,"** describes the Converter export file used to link code against another package.

**Chapter 6, "The Cap File Format,"** describes the format of the .cap file.

**Chapter 7, "Instruction Set,"** describes the byte codes (opcodes) that comprise the Java Card Virtual Machine instruction set.

**Chapter 8, "Tables of Instructions,"** summarizes the Java Card VM instructions in two different tables: one sorted by Opcode Value and the other sorted by Mnemonic.

**Glossary** is a list of words and their definitions to assist you in using this book.

---

## Prerequisites

This specification is not intended to stand on its own; rather it relies heavily on existing documentation of the Java platform. In particular, two books are required for the reader to understand the material presented here.

[1] Gosling, James, Bill Joy, and Guy Steele. *The Java™ Language Specification*. Addison-Wesley, 1996, ISBN 0-201-63451-1 – contains the definitive definition of the Java programming language. The Java Card 2.1 language subset defined here is based on the language specified in this book.

[2] Lindholm, Tim, and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, 1996, ISBN 0-201-63452-X – defines the standard operation of the Java Virtual Machine. The Java Card virtual machine presented here is based on the definition specified in this book.

---

## Related Documents

References to various documents or products are made in this manual. You should have the following documents available:

- *Java Card 2.1 Application Programming Interface*, Draft 2 Revision 1.4, Sun Microsystems, Inc.
- *Java Card Runtime Environment (JCRE) 2.1 Specification* Draft 1 Revision 1.2, Sun Microsystems, Inc.
- *Java Card Applet Developer's Guide*, Sun Microsystems, Inc.
- *The Java Language Specification* by James Gosling, Bill Joy, and Guy L. Steele. Addison-Wesley, 1996, ISBN 0-201-63451-1.
- *The Java Virtual Machine Specification (Java Series)* by Tim Lindholm and Frank Yellin. Addison-Wesley, 1996, ISBN 0-201-63452-X.
- *The Java Class Libraries: An Annotated Reference (Java Series)* by Patrick Chan and Rosanna Lee. Addison-Wesley, ISBN: 0201634589.
- ISO 7816 Specification Parts 1-6.
- EMV '96 Integrated Circuit Card Specification for Payment Systems.

---

## Ordering Sun Documents

The SunDocs<sup>SM</sup> program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress<sup>TM</sup> Internet site at <http://www.sun.com/sunexpress>.

---

## What Typographic Changes Mean

The following table describes the typographic changes used in this book.

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name%</code> You have mail.
AaBbCc123	What you type, contrasted with on-screen computer output	<code>machine_name%</code> <code>su</code> Password:
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type <code>rm filename</code> .
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> be root to do this.

# Introduction

---

---

## 1.1 Motivation

Java Card technology enables programs written in the Java programming language to be run on smart cards and other small, resource-constrained devices. Developers can build and test programs using standard software development tools and environments, then convert them into a form that can be installed onto a Java Card enabled device. Application software for the Java Card platform is called an applet, or more specifically, a Java Card applet or card applet (to distinguish it from browser applets).

While Java Card technology enables programs written in the Java programming language to run on smart cards, such small devices are far too under-powered to support the full functionality of the Java platform. Therefore, the Java Card platform supports only a carefully chosen, customized subset of the features of the Java platform. This subset provides features that are well-suited for writing programs for small devices, and preserves most of the “feel” and all of the object-oriented capabilities of the Java programming language.

A simple approach to specifying a Java Card Virtual Machine would be to describe the subset of the features of the Java Virtual Machine that must be supported to allow for portability of source code across all Java Card enabled devices. Combining that subset specification and the information in the Java Virtual Machine Specification, smart card manufacturers could construct their own Java Card implementations. While that approach is feasible, it has a serious drawback. The resultant platform would be missing the important feature of binary portability of Java Card applets.

The standards that define the Java platform allow for binary portability of Java programs across all Java platform implementations. This “write once, run anywhere” quality of Java programs is perhaps the most significant feature of the platform. Part of the motivation for the creation of the Java Card platform was to bring just this kind of binary portability to the

smart card industry. In a world with hundreds of millions or perhaps even billions of smart cards with varying processors and configurations, the costs of supporting multiple binary formats for software distribution could be overwhelming.

This Java Card Virtual Machine Specification is the key to providing binary portability. One way of understanding what this specification does is to compare it to its counterpart in the Java platform. The Java Virtual Machine Specification defines a Java Virtual Machine as an engine that loads Java `class` files and executes them with a particular set of semantics. The `class` file is a central piece of the Java architecture, and it is the standard for the binary compatibility of the Java platform. The Java Card Virtual Machine specification also defines a file format that is the standard for binary compatibility for the Java Card platform: the CAP file format is the form in which software is loaded onto Java Card enabled devices.

## 1.2 The Java Card Virtual Machine

The role of the Java Card Virtual Machine is best understood in the context of the process for production and deployment of Java Card software. There are several components that make up a Java Card system, including the Java Card Virtual Machine, the Java Card Converter, an off-card installation tool, and an on-card installation program, as shown in Figures 1-1 and 1-2.

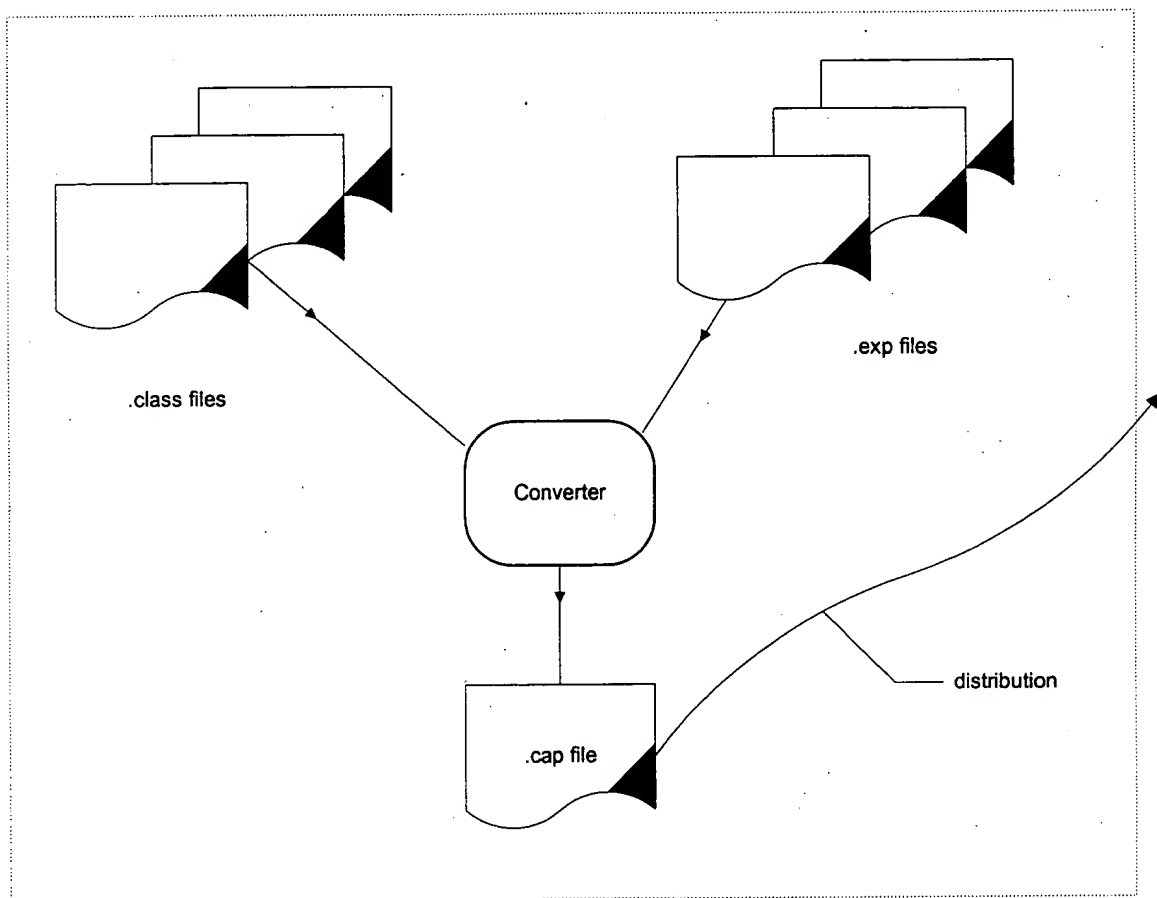
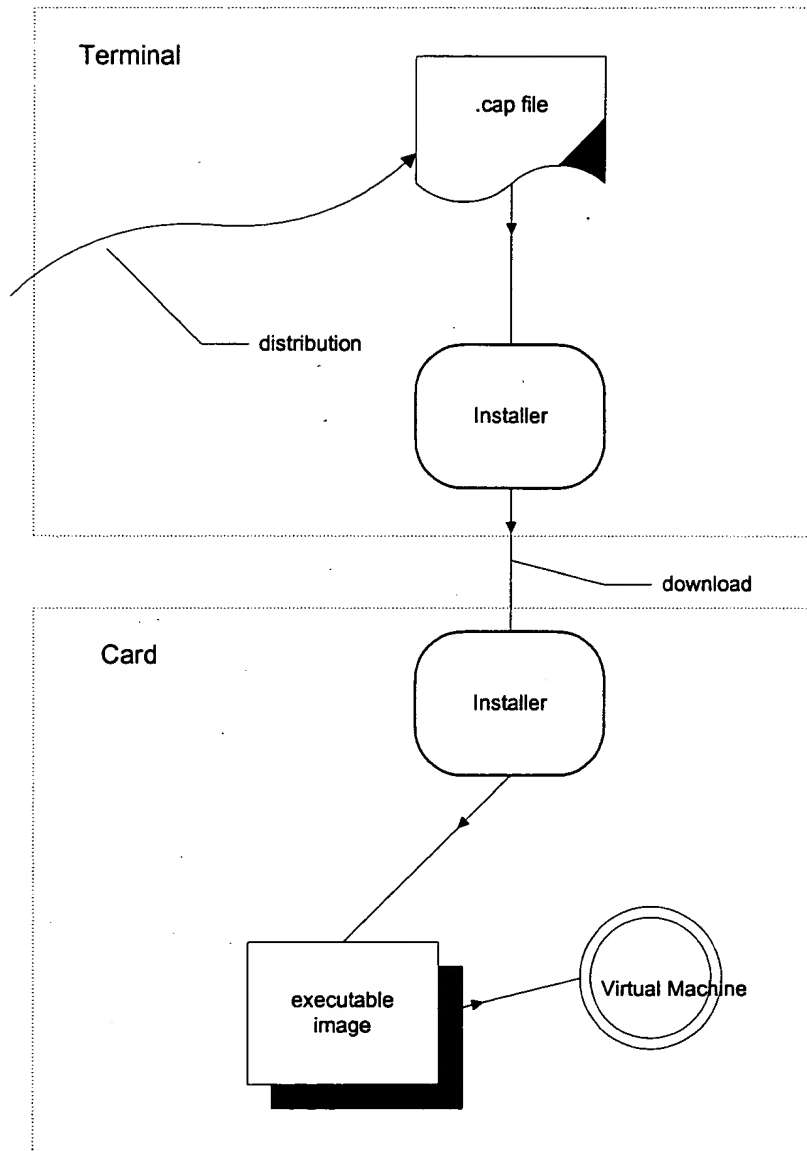


FIGURE 1-1 Java Card Applet Conversion and Distribution



**FIGURE 1-2** Java Card Applet Distribution and Installation

Development of a Java Card applet begins as with any other Java program: a developer writes one or more Java classes, and compiles the source code with a Java compiler, producing one or more `class` files. The applet is run, tested and debugged on a workstation using simulation tools to emulate the on-card environment. Then, when an applet is ready to

be downloaded to a Java Card enabled device, the `class` files comprising the applet are converted to a CAP (converted applet) file using a Java Card Converter. The Java Card Converter takes as input not only the `class` files to be converted, but also one or more *export files*. An export file contains naming information for the contents of other packages that are imported by the classes being converted.

After conversion, the CAP file is copied to a card terminal, such as a desktop computer with a card reader peripheral. Then an installation tool on the terminal loads the CAP file and transmits it to the Java Card enabled device. An installation program on the Java Card enabled device receives the contents of the CAP file and prepares the applet to be run by the Java Card Virtual Machine. The virtual machine itself need not load or manipulate CAP files; it need only execute the applet code found in the CAP file that was loaded onto the card by the on-card installation program.

The division of functionality between the Java Card Virtual Machine and the on-card installation program keeps both the virtual machine and the installation program small. The installation program can be implemented as a Java program and executed on top of the Java Card Virtual Machine. Since Java Card instructions are denser than typical machine code, this reduces the size of the installer. The modularity also allows different installers to be used with a single Java Card Virtual Machine implementation.

---

## 1.3 Java Language Security

One of the fundamental features of the Java Virtual Machine is the strong security provided in part by the `class` file verifier. Many Java Card enabled devices may be too small to support on-card verification of CAP files. This consideration led to a design that enables on-card verification but does not rely on it. The data in a CAP file that is needed only for verification is packaged separately from the data needed for the actual execution of its applet. This allows for flexibility in how security is managed on a Java Card enabled device.

There are several options for providing language-level security on a Java Card enabled device. The conceptually simplest is to verify the contents of a CAP file on the card as it is downloaded or after it is downloaded. This option might only be feasible in the largest of Java Card enabled devices. However, some subset of verification might be possible even on smaller devices. Other options rely on some combination of one or more of: physical security of the installation terminal, a cryptographically enforced chain of trust from the source of the CAP file, and off-card verification of the contents of a CAP file.

The Java Card platform standards say as little as possible about CAP file installation and security policies. Since smart cards must serve as secure processors in many different systems with different security requirements, it is necessary to allow a great deal of flexibility to meet the needs of smart card issuers and users.

---

## 1.4 Java Card Runtime Environment Security

The standard runtime environment for the Java Card platform is the Java Card Runtime Environment (JCRE). The JCRE consists of the Java Card Virtual Machine along with the Java Card API classes. While the Java Card Virtual Machine has responsibility for ensuring Java language-level security, the JCRE imposes additional runtime security requirements on Java Card enabled devices which implement the JCRE, which results in a need for additional features on the Java Card Virtual Machine. Throughout this document, these additional features are designated as JCRE-specific, as shown in the example below.

The basic runtime security feature imposed by the JCRE enforces isolation of applets using what is called an *applet firewall*. The applet firewall prevents the objects that were created by one applet from being used by another applet. This prevents unauthorized access to both the fields and methods of class instances, as well as the length and contents of arrays.

Isolation of applets is an important security feature, but it requires a mechanism to allow applets to share objects in situations where there is a need to interoperate. The JCRE allows such sharing using the concept of *sharable interface objects*. These objects are the only way an applet can make its objects available for use by other applets. For more information about using sharable interface objects, see the description of the `javacard.framework.Sharable` interface in the Java Card API Specification. Some descriptions of firewall-related features will make reference to the `Sharable` interface.

The applet firewall also protects from unauthorized use the objects owned by the JCRE itself. The JCRE can use mechanisms not reflected in the Java Card API to make its objects available for use by applets. A full description of the JCRE-related isolation and sharing features can be found in the *Java Card Runtime Environment Specification*.

## A Subset of the Java Virtual Machine

---

This chapter describes the subset of the Java virtual machine and language that is supported in the Java Card 2.1 platform.

---

### 2.1 Why a Subset is Needed

It would be ideal if programs for smart cards could be written using all of the Java programming language, but a full implementation of the Java virtual machine is far too large to fit on even the most advanced memory-constrained devices available today.

A typical memory-constrained device has less than 1K of RAM and 16K of ROM. The code for implementing string manipulation, single and double-precision floating point arithmetic, and thread management would be larger than the ROM space on such a device. Even if it could be made to fit, there would be no space left over for class libraries or application code. RAM resources are also very limited. The only workable option is to implement Java Card as a subset of the Java platform.

### 2.2 Java Card Language Subset

Implementations and applets written for the Java Card platform are written in the Java programming language. They are compiled using Java compilers. Java Card technology uses a subset of the Java language, and familiarity with the Java platform is required to understand the Java Card platform.

The items discussed in this section are not described to the level of a language specification. For complete documentation on the Java programming language, *The Java Language Specification*. (§1.1)

## 2.2.1 Unsupported Items

The items listed in this section are elements of the Java programming language and platform that are not supported by the Java Card platform.

### 2.2.1.1 Unsupported Features

#### *Dynamic Class Loading*

Dynamic class loading is not supported in the Java Card platform. An implementation of the Java Card platform is not able to load classes dynamically. Classes are either masked into the card during manufacturing or downloaded through an installation process after the card has been issued. Programs executing on the card may only refer to classes that already exist on the card, since there is no way to download classes during the normal execution of application code.

#### *Security Manager*

Security management in the Java Card platform differs significantly from that of the Java platform. In the Java platform, there is a Security Manager class (`java.lang.SecurityManager`) responsible for implementing security features. In the Java Card platform, language security policies are implemented by the Virtual Machine. There is no Security Manager class that makes policy decisions on whether to allow operations.

#### *Garbage Collection & Finalization*

Java Card does not require a garbage collector. Nor does Java Card allow explicit deallocation of objects, since this would break Java programming language's required pointer-safety. Therefore, application programmers may not assume that objects that are allocated are ever deallocated. Storage for unreachable objects will not necessarily be reclaimed.

Finalization is also not required. `finalize()` will not necessarily be called automatically by the Java Card virtual machine, and programmers should not rely on this behavior.

#### *Threads*

The Java Card virtual machine does not support multiple threads of control. Neither class `Thread` nor any of the thread-related keywords can be used in implementations of the Java Card platform.

### *Cloning*

The Java Card platform does not support cloning of objects. Java Card API class `Object` does not implement a `clone()` method, and there is no `Cloneable` interface provided.

### *Access Control in Java Packages*

The Java Card language subset supports the package access control defined in the Java language. However, there are two cases that are not supported.

- If a class implements a method with package access visibility, a subclass may not override the method and change the access visibility of the method to `protected` or `public`.
- An interface that is defined with package access visibility may not be extended by an interface with `public` access visibility.

#### 2.2.1.2 Keywords

The following keywords indicate unsupported options related to threads or memory management.

`synchronized`      `transient`      `volatile`

#### 2.2.1.3 Unsupported Types

The Java Card platform does not support types `char`, `double`, `float` or `long`, or operations on those types. It also does not support arrays with more than one dimension.

#### 2.2.1.4 Classes

In general, none of the Java classes are supported in the Java Card platform. Some classes from the `java.lang` package are supported (see Section 2.2.2.4, “Classes”), but none of the rest are. For example, classes that are *not* supported are `String`, `Thread` (and all thread-related classes), wrapper classes such as `Boolean` and `Integer`, and class `Class`.

### *System*

Class `java.lang.System` is not supported. Java Card supplies a class `javacard.framework.JCSystem`, which provides an interface to system behavior.

## 2.2.2 Supported Items

If a language feature is not explicitly described as unsupported, it is part of the supported subset. Notable supported features are described in this section.

### 2.2.2.1 Features

#### *Packages*

Implementations and applets written for the Java Card platform follow the standard rules for the Java platform packages. Java Card API classes are written as java source files, which include package designations. Package mechanisms are used to identify and control access to classes, static fields and static methods. Except as noted in "Access Control in Java Packages" on page 9, packages in the Java Card platform are used exactly the way they are in the Java platform.

#### *Dynamic Object Creation*

The Java Card platform programs supports dynamically created objects, both class instances and arrays. This is done, as usual, by using the `new` operator. Objects are allocated out of the heap.

As noted in "Garbage Collection & Finalization" on page 8, a Java Card Virtual Machine will not necessarily garbage collect objects. Any objects allocated on the card may continue to exist and consume resources even after they become unreachable.

#### *Virtual Methods*

Since Java Card objects are Java programming language objects, invoking virtual methods on objects in a program written for the Java Card platform is exactly the same as in a program written for the Java platform. Inheritance is supported, including the use of the `super` keyword.

#### *Interfaces*

Java Card classes may define or implement Interfaces as in the Java programming language. Invoking virtual methods on interface types works as expected. Type checking and the `instanceof` operator also work correctly with interfaces.

## *Exceptions*

Java Card programs may define, throw and catch exceptions, as in Java programs. Class `Throwable` and its relevant subclasses are supported. (Some `Exception` and `Error` subclasses are omitted, since those exceptions cannot occur in the Java Card platform. See Section 2.3.3, “Exceptions” for specification of errors and exceptions.)

### 2.2.2.2 Keywords

The following keywords are supported. Their use is the same as in the Java programming language.

<code>abstract</code>	<code>default</code>	<code>If</code>	<code>package</code>	<code>switch</code>
<code>boolean</code>	<code>do</code>	<code>implements</code>	<code>private</code>	<code>this</code>
<code>break</code>	<code>else</code>	<code>import</code>	<code>protected</code>	<code>throw</code>
<code>byte</code>	<code>extends</code>	<code>instanceof</code>	<code>public</code>	<code>throws</code>
<code>case</code>	<code>final</code>	<code>int</code>	<code>return</code>	<code>try</code>
<code>catch</code>	<code>finally</code>	<code>interface</code>	<code>short</code>	<code>void</code>
<code>class</code>	<code>for</code>	<code>native</code>	<code>static</code>	<code>while</code>
<code>continue</code>	<code>goto</code>	<code>new</code>	<code>super</code>	

### 2.2.2.3 Types

Java programming language types `boolean`, `byte`, `short`, and `int` are supported. Objects (class instances and single-dimensional arrays) are also supported. Arrays can contain the supported primitive data types, objects, and other arrays.

Some Java Card implementations might not support use of the `int` data type. (Refer to Section 2.2.3.1, “`int`.”)

### 2.2.2.4 Classes

Most of the classes in the `java.lang` package are not supported in Java Card. The following classes from `java.lang` are supported on the card in a limited form.

#### *Object*

Java Card classes descend from `java.lang.Object`, just as in the Java programming language. Most of the methods of `Object` are not available in the Java Card API, but the class itself exists to provide a root for the class hierarchy.

### *Throwable*

Class `Throwable` and its subclasses are supported. Most of the methods of `Throwable` are not available in the Java Card API, but the class itself exists to provide a common ancestor for all exceptions.

## 2.2.3 Optionally Supported Items

This section describes what items are optionally supported. Optional means that a Java Card implementation is not required to support the specified items.

Optional features of the Java programming language are optional in the Java Card platform or not required for a Java Card compatible implementation. These features are described below.

### 2.2.3.1 `int`

The `int` keyword and 32-bit integer data types need not be supported in a Java Card implementation. A Java Card Virtual Machine that does not support the `int` data type will reject programs using that type.

## 2.2.4 Limitations of the Java Card Virtual Machine

The limitations of card hardware prevent Java Card programs from supporting the full range of functionality of certain Java platform features. The features in question are supported, but a particular virtual machine may limit the range of operation to less than that of the Java platform.

To ensure a level of portability for application code, this section establishes a minimum required level for partial support of these language features.

The limitations here are listed as maximums from the application programmer's perspective. Applets that do not violate these maximum values will be portable across all Java Card implementations. From the Java Card VM implementer's perspective, each maximum listed indicates a minimum level of support that will allow portability of applets.

#### 2.2.4.1 Objects

##### *Methods*

A class can implement a maximum of 128 public or protected instance methods, and a maximum of 128 instance methods with package visibility. These limits include inherited methods.

##### *Class Instances*

Class instances can contain a maximum of 256 fields, where an `int` data type is counted as occupying two fields.

##### *Arrays*

Arrays can hold a maximum of 32767 fields.

#### 2.2.4.2 Methods

The maximum size of a stack frame is 127 words, where an `int` data type is counted as occupying two words. This includes the parameters, locals, and operand stack.

#### 2.2.4.3 Switch Statements

Java Card programs are limited to a maximum of 65536 cases in a switch statement.

#### 2.2.4.4 Class Initialization

There is limited support for initialization of static field values in `<clinit>` methods. Static fields of applets may only be initialized to primitive compile-time constant values, or arrays of primitive compile-time constants. Static fields of user libraries may only be initialized to primitive compile-time constant values. Primitive constant data types include `boolean`, `byte`, `short`, and `int`.

---

## 2.3 Java Card VM Subset

Java Card technology uses a subset of the Java Virtual Machine, and familiarity with the Java platform is required to understand the Java Card Virtual Machine.

The items discussed in this section are not described to the level of a virtual machine specification. For complete documentation on the Java Virtual Machine, refer to §1.1 of *The Java™ Virtual Machine Specification*.

### 2.3.1 class File Subset

The Java Card Virtual Machine operates on standard Java platform `class` files. Since the Java Card Virtual Machine supports only a subset of the behavior of the Java Virtual Machine, it also supports only a subset of the standard `class` file format.

#### 2.3.1.1 Not Supported in Class Files

##### *Field Descriptors*

Field descriptors may not contain *BaseType* characters **C**, **D**, **F** or **L**. *ArrayType* descriptors for arrays of more than one dimension may not be used.

##### *Constant Pool*

Constant pool table entry tags that indicate unsupported types are not supported.

**TABLE 2-1** Unsupported constant pool tags

Constant Type	Value
CONSTANT_String	8
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6

Constant pool structures for types `CONSTANT_String_info`, `CONSTANT_Float_info`, `CONSTANT_Long_info` and `CONSTANT_Double_info` are not supported.

### *Fields*

In `field_info` structures, the access flags `ACC_VOLATILE` and `ACC_TRANSIENT` are not supported.

### *Methods*

In `method_info` structures, the access flags `ACC_SYNCHRONIZED` and `ACC_NATIVE` are not supported.

## 2.3.1.2 Supported in Class Files

### *ClassFile*

All items in the `ClassFile` structure are supported.

### *Field Descriptors*

Field descriptors may contain *BaseType* characters **B**, **I**, **S** and **Z**, as well as any *ObjectType*. *ArrayType* descriptors for arrays of a single dimension may also be used.

### *Method Descriptors*

All forms of method descriptors are supported.

### *Constant pool*

Constant pool table entry tags for supported data types are supported.

**TABLE 2-2** Supported constant pool tags.

Constant Type	Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_Integer	3
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

Constant pool structures for types `CONSTANT_Class_info`, `CONSTANT_Fieldref_info`, `CONSTANT_Methodref_info`, `CONSTANT_InterfaceMethodref_info`, `CONSTANT_Integer_info`, `CONSTANT_NameAndType_info` and `CONSTANT_Utf8_info` are supported.

### *Fields*

In `field_info` structures, the supported access flags are `ACC_PUBLIC`, `ACC_PRIVATE`, `ACC_PROTECTED`, `ACC_STATIC` and `ACC_FINAL`.

The remaining components of `field_info` structures are fully supported.

### *Methods*

In `method_info` structures, the supported access flags are `ACC_PUBLIC`, `ACC_PRIVATE`, `ACC_PROTECTED`, `ACC_STATIC`, `ACC_FINAL` and `ACC_ABSTRACT`. The access flag `ACC_NATIVE` is supported for non-applet class files.

The remaining components of `method_info` structures are fully supported.

### *Attributes*

The `attribute_info` structure is supported. The `Code`, `ConstantValue`, `Exceptions` and `LocalVariableTable` attributes are supported.

## 2.3.2 Bytecode Subset

The following sections detail the bytecodes that are either supported or unsupported in the Java Card platform. For more details, refer to Chapter 6, "Instruction Set."

### 2.3.2.1 Unsupported Bytecodes

lconst_<l>	fconst_<f>	dconst_<d>	ldc2_w2
lload	fload	dload	lload_<n>
fload_<n>	dload_<n>	laload	faload
daload	caload	lstore	fstore
dstore	lstore_<n>	fstore_<n>	dstore_<n>
lastore	fastore	dastore	castore
ladd	fadd	dadd	lsub
fsub	dsub	lmul	fmul
dmul	ldiv	fdiv	ddiv
lrem	frem	drem	lneg
fneg	dneg	lshl	lshr
lushr	land	lor	lxor
i2l	i2f	i2d	l2i
l2f	l2d	f2i	f2d
d2i	d2l	d2f	i2c
lcmp	fcmpl	fcmpg	dcmpl
dcmpg	lreturn	freturn	dreturn
monitorenter	monitorexit	multianewarray	goto_w
jsr_w			

### 2.3.2.2 Supported Bytecodes

nop	aconst_null	iconst_<i>	bipush
sipush	ldc	ldc_w	iload
aload	iload_<n>	aload_<n>	iaload
aaload	baload	saload	istore
astore	istore_<n>	astore_<n>	iastore
aastore	bastore	sastore	pop
pop2	dup	dup_x1	dup_x2
dup2	dup2_x1	dup2_x2	swap
iadd	isub	imul	idiv
irem	ineg	ior	ishl
ishr	iushr	iand	ixor
iinc	i2b	i2s	if<cond>
ificmp_<cond>	ifacmp_<cond>	goto	jsr
ret	tableswitch	lookupswitch	ireturn
areturn	return	getstatic	putstatic
getfield	putfield	invokevirtual	invokespecial

<code>invokestatic</code>	<code>invokeinterface</code>	<code>new</code>	<code>newarray</code>
<code>anewarray</code>	<code>arraylength</code>	<code>athrow</code>	<code>checkcast</code>
<code>instanceof</code>	<code>wide</code>	<code>ifnull</code>	<code>ifnonnull</code>

### 2.3.2.3 Static Restrictions on Bytecodes

For it to be acceptable to a Java Card Virtual Machine, a `class` file must conform to the following restrictions on the static form of bytecodes.

#### *ldc, ldc\_w*

The `ldc` and `ldc_w` bytecodes can only be used to load integer constants. The constant pool entry at *index* must be a `CONSTANT_Integer` entry.

#### *lookupswitch*

The value of the *npairs* operand must be less than 65536. The bytecode can contain at most 65535 cases.

#### *tableswitch*

The values of the *high* and *low* operands must both be less than 65536 (so they can fit in 16 bits). The bytecode can contain at most 65535 cases.

#### *wide*

The *wide* bytecode cannot be used to generate local indices greater than 127, and it cannot be used with any instructions other than `iinc`. It can only be used with an `iinc` bytecode to extend the range of the increment constant.

## 2.3.3 Exceptions

Java Card provides full support for the Java platform's exception mechanism. Users can define, throw and catch exceptions just as in the Java platform. Java Card also makes use of the exceptions and errors defined in *The Java Language Specification* [1]. An updated list of the Java platform's exceptions is provided in the JDK documentation.

Not all of the Java platform's exceptions are supported in Java Card. Exceptions related to unsupported features are naturally not supported. Class loader exceptions (the bulk of the checked exceptions) are not supported. And no exceptions or errors defined in packages other than `java.lang` are supported.

Note that some exceptions may be supported to the extent that their error conditions are detected correctly, but classes for those exceptions will not necessarily be present in the API.

The supported subset is described in the tables below.

### 2.3.3.1 Uncaught and Uncatchable Exceptions

In the Java platform, uncaught exceptions and errors will cause the virtual machine to report the error condition and exit. In Java Card, uncaught exceptions or errors should cause the Virtual Machine to halt. Optionally, an implementation can require the card to be muted or blocked from further use.

Throwing a runtime exception or error that cannot be caught should also cause the card to be muted. Cards may also optionally take stricter actions in response to throwing such an exception.

### 2.3.3.2 Checked Exceptions

TABLE 2-3 Support of checked exceptions

Exception	Supported	Not Supported
<code>ClassNotFoundException</code>		•
<code>CloneNotSupportedException</code>		•
<code>IllegalAccessException</code>		•
<code>InstantiationException</code>		•
<code>InterruptedException</code>		•
<code>NoSuchFieldException</code>		•
<code>NoSuchMethodException</code>		•

### 2.3.3.3

## Runtime Exceptions

TABLE 2-4 Support of runtime exceptions

Runtime Exception	Supported	Not Supported
ArithmeticException	•	
ArrayStoreException	•	
ClassCastException	•	
IllegalArgumentException	•	
IllegalThreadStateException		•
NumberFormatException		•
IllegalMonitorStateException		•
IllegalStateException	•	
IndexOutOfBoundsException	•	
ArrayIndexOutOfBoundsException	•	
StringIndexOutOfBoundsException		•
NegativeArraySizeException	•	
NullPointerException	•	
SecurityException	•	

### 2.3.3.4

## Errors

TABLE 2-5 Support of errors

Error	Supported	Not Supported
LinkageError	•	
ClassCircularityError	•	
ClassFormatError	•	
ExceptionInInitializerError	•	
IncompatibleClassChangeError	•	
AbstractMethodError	•	
IllegalAccessError	•	
InstantiationError	•	
NoSuchFieldError	•	
NoSuchMethodError	•	
NoClassDefFoundError	•	
UnsatisfiedLinkError	•	

Error	Supported	Not Supported
VerifyError	•	
ThreadDeath		•
VirtualMachineError	•	
InternalError	•	
OutOfMemoryError	•	
StackOverflowError	•	
UnknownError	•	

*This Page Blank (uspio)*

## Structure of the Java Card Virtual Machine

---

The specification of the Java Card Virtual Machine is in many ways quite similar to that of the Java Virtual Machine. This similarity is of course intentional, as the design of the Java Card Virtual Machine was based on that of the Java Virtual Machine. Rather than reiterate all the details of this specification which are shared with that of the Java Virtual Machine, this chapter will mainly refer to its counterpart in the *Java Virtual Machine Specification*, providing new information only where the Java Card Virtual Machine differs.

---

### 3.1 Data Types and Values

The Java Card Virtual Machine supports the same two kinds of data types as the Java Virtual Machine: *primitive types* and *reference types*. Likewise, the same two kinds of values are used: *primitive values* and *reference values*.

The primitive data types supported by the Java Card Virtual Machine are the *numeric types* and the *returnAddress* type. The numeric types consist only of the *integral types*:

- *byte*, whose values are 8-bit signed two's complement integers
- *short*, whose values are 16-bit signed two's complement integers

Some Java Card Virtual Machine implementations may also support an additional integral type:

- *int*, whose values are 32-bit signed two's complement integers

Support for reference types is identical to that in the Java Virtual Machine.

---

## 3.2 Words

The Java Card Virtual Machine is defined in terms of an abstract storage unit called a *word*. This specification does not mandate the actual size in bits of a word on a specific platform. A word is large enough to hold a value of type `byte`, `short`, `reference` or `returnAddress`. Two words are large enough to hold a value of type `int`.

The actual storage used for values in an implementation is platform-specific. There is enough information present in the descriptor component of a CAP file to allow an implementation to optimize the storage used for values in variables and on the stack.

---

## 3.3 Runtime Data Areas

The Java Card Virtual Machine can support only a single thread of execution. Any runtime data area in the Java Virtual Machine which is duplicated on a per-thread basis will have only one global copy in the Java Card Virtual Machine.

The Java Card Virtual Machine's heap is not required to be garbage collected. Objects allocated from the heap will not necessarily be reclaimed.

This specification does not include support for native methods, so there are no native method stacks.

Otherwise, the runtime data areas are as documented for the Java Virtual Machine.

---

## 3.4 Applet Contexts

Each applet running on a Java Card Virtual Machine is associated with an execution context known as an *applet context*. The Java Card Virtual Machine uses the applet context of the current frame to enforce security policies for inter-applet operations.

There is a one-to-one mapping between applet contexts and packages in which applets are defined. An easy way to think of an applet context is as the runtime equivalent of a package, since Java packages are compile-time constructs and have no direct representation at runtime. As a consequence, all applets managed by applet instances of applet classes from the same package will share the same applet context.

The Java Card Runtime Environment also has its own applet context. Framework objects execute in this *JCRE context*.

The applet context of the currently executing method is known as the *current context*. When a method in one context successfully invokes a method in another context, the Java Card Virtual Machine performs an *applet context switch*. Afterwards the invoked method's applet context becomes the current context. When the invoked method returns, the current context is switched back to the previous context.

---

## 3.5 Frames

Java Card Virtual Machine *frames* are very similar to those defined for the Java Virtual Machine. Each frame has a set of local variables and an operand stack. Frames also contain a reference to a constant pool, but since all constant pools for all classes in a package are merged, the reference is to the constant pool for the current class' package.

Each frame also includes a reference to the applet context in which the current method is executing.

---

## 3.6 Representation of Objects

The Java Card Virtual Machine does not mandate a particular internal structure for objects or a particular layout of their contents. However, the core components in a CAP file are defined assuming a default structure for certain runtime structures (such as descriptions of classes), and a default layout for the contents of dynamically allocated objects. Information from the descriptor component of the CAP file can be used to format objects in whatever way an implementation requires.

---

## 3.7 Special Initialization Methods

The Java Card Virtual Machine supports *instance initialization methods* exactly as does the Java Virtual Machine.

The Java Card Virtual Machine includes only limited support for *class or interface initialization methods*. There is no general mechanism for executing `<clinit>` methods on a Java Card Virtual Machine. Instead, a CAP file includes information for initializing class data as defined in Chapter 2, "A Subset of the Java Virtual Machine."

---

## 3.8 Exceptions

Exception support in the Java Card Virtual Machine is identical to support for exceptions in the Java Virtual Machine.

---

## 3.9 Binary File Formats

This specification defines two binary file formats which enable platform-independent development, distribution and execution of Java Card software.

The CAP file format describes files that contain executable code and can be downloaded and installed onto a Java Card enabled device. A CAP file is produced by a Java Card Converter tool, and contains a converted form of an entire package of Java classes. This file format's relationship to the Java Card Virtual Machine is analogous to the relationship of the `class` file format to the Java Virtual Machine.

The export file format describes files that contain the public linking information of Java Card packages. A package's export file is used when converting client packages of that package.

---

## 3.10 Instruction Set Summary

The Java Card Virtual Machine instruction set is quite similar to the Java Virtual Machine instruction set. Individual instructions consist of a one-byte *opcode* and zero or more *operands*. The pseudo-code for the Java Card Virtual Machine's instruction fetch-decode-execute loop is the same. And multi-byte operand data is also encoded in *big-endian* order.

There are a number of ways in which the Java Card Virtual Machine instruction set diverges from that of the Java Virtual Machine. Most of the differences are due to the Java Card Virtual Machine's more limited support for data types. Another source of divergence is that the Java Card Virtual Machine is intended to run on 8-bit and 16-bit architectures, whereas the Java Virtual Machine was designed for a 32-bit architecture. The rest of the differences are all oriented in one way or another toward optimizing the size or performance of either the Java Card Virtual Machine or Java Card programs. These changes include inlining constant pool data directly in instruction opcodes or operands, adding multiple versions of a particular instruction to deal with different datatypes, and creating composite instructions for operations on the current object.

### 3.10.1 Types and the Java Card Virtual Machine

The Java Card Virtual Machine supports only a subset of the types supported by the Java Virtual Machine. This subset is described in Chapter 2, "A Subset of the Java Virtual Machine." Type support is reflected in the instruction set, as instructions encode the data types on which they operate.

Given that the Java Card Virtual Machine supports fewer types than the Java Virtual Machine, there is an opportunity for better support for smaller data types. Lack of support for large numeric data types frees up space in the instruction set. This extra instruction space has been used to directly support arithmetic operations on the short data type.

Some of the extra instruction space has also been used to optimize common operations. Type information is directly encoded in field access instructions, rather than being obtained from an entry in the constant pool.

TABLE 3-1 summarizes the type support in the instruction set of the Java Card Virtual Machine. Only instructions that exist for multiple types are listed. Wide and composite forms of instructions are not listed either. A specific instruction, with type information, is built by replacing the *T* in the instruction template in the opcode column by the letter representing the type in the type column. If the type column for some instruction is blank, then no instruction exists supporting that operation on that type. For instance, there is a load instruction for type short, *sload*, but there is no load instruction for type byte.

TABLE 3-1 Type support in the Java Card Virtual Machine Instruction Set

opcode	byte	short	int	reference
Tspush	bspush	sspush		
Tipush	bipush	sipush	iipush	
Tconst		sconst	iconst	aconst
Tload		sload	iload	aload
Tstore		sstore	istore	astore
Tinc		sinc	iinc	
Taload	baload	saload	iaload	aaload
Tastore	bastore	sastore	iastore	aastore
Tadd		sadd	iadd	
Tsub		ssub	isub	
Tmul		smul	imul	
Tdiv		sdiv	idiv	
Trem		srem	irem	
Tneg		sneg	ineg	

**TABLE 3-1** Type support in the Java Card Virtual Machine Instruction Set

opcode	byte	short	int	reference
Tshl		sshl	ishl	
Tshr		sshr	ishr	
Tushr		sushr	iushr	
Tand		sand	iand	
Tor		sor	ior	
Txor		sxor	icor	
s2T	s2b		s2i	
i2T	i2b	i2s		
Tcmp			icmp	
if_TcmpOP		if_scmpOP		if_acmpOP
Tlookupswitch		slookupswitch	ilookupswitch	
Ttableswitch		stableswitch	itableswitch	
Treturn		sreturn	ireturn	areturn
getstatic_T	getstatic_b	getstatic_s	getstatic_i	getstatic_a
putstatic_T	putstatic_b	putstatic_s	putstatic_i	putstatic_a
getfield_T	getfield_b	getfield_s	getfield_i	getfield_a
putfield_T	putfield_b	putfield_s	putfield_i	putfield_a

The mapping between Java storage types and Java Card Virtual Machine computational types is summarized by the following table:

**TABLE 3-2** Storage types and computational types

Java (Storage) Type	Size in Bits	Computational Type
byte	8	short
short	16	short
int	32	int

Chapter 7, “Java Card Virtual Machine Instruction Set,” describes the Java Card Virtual Machine instruction set in detail.

## Java Card Naming

---

This chapter describes the mechanisms used for naming items in Java Card CAP files and Export files. Java class files use Unicode strings organized with a particular convention to name Java packages as well as items within those packages. As the Java Card platform does not include support for strings, alternative support for naming is provided.

This chapter describes a scheme that allows downloaded software to be linked against APIs on a Java Card enabled smart card. The scheme represents referenced items as opaque tokens, instead of Unicode strings as are used in Java class files. The two basic requirements of this linking scheme are that it allows linking on the card, and that it does not require internal implementation details of card APIs to be revealed to clients of those APIs. Secondary requirements are that the scheme be efficient in terms of resource use on the card, and have good performance for linking. And of course, it must preserve the semantics of the Java language.

---

### 4.1 Overview of Token-based Linking

This section provides an overview of tokens and their roles in the various stages of the production and installation of packages which implement applets and user libraries.

#### 4.1.1 Externally Visible Items

Classes (including Interfaces) in Java packages may be declared with public or package visibility. A class's methods and fields may be declared with public, protected, package or private visibility. For purposes of this document, we define public classes, public or protected fields, and public or protected methods to be *externally visible* from the package. All externally visible items are described in a package's Export file.

Each externally visible item must have a token associated with it to enable references from other packages to the item to be resolved on-card. There are six kinds of items in a package that require external identification.

- n Classes (including Interfaces)
- n Static Fields
- n Static Methods
- n Instance Fields
- n Virtual Methods
- n Interface Methods

### 4.1.2 Private Tokens

Items which are not externally visible are *internally visible*. Internally visible items are not described in a package's export file, but some such items use *private tokens* to represent internal references. External references are represented by *public tokens*. There are two kinds of items which can be assigned private tokens.

- n Instance Fields
- n Virtual Methods

### 4.1.3 The Export File and Conversion

Each externally visible item in a package has an entry in the package's export file. Each entry holds the item's name and its token. Some entries may include additional information as well. For detailed information on the export file format, see Chapter 5, "The Export File Format."

The export file is used to map names for imported items to tokens during package conversion.

During the conversion of the class files of applet A, the export file of `javacard.framework` is used to find tokens for items in the API which are used by the applet. The Java Card converter uses these tokens to represent references to items in the API.

For instance, an applet creates a new instance of framework class PIN. The framework export file contains an entry for `javacard.framework.PIN` which holds the token for this class. The converter places this token in the CAP file's constant pool to represent an unresolved reference to the class. The token value is used to resolve the reference on a card.

#### 4.1.4 References – External and Internal

In the context of a CAP file, references to items are made indirectly through a package's constant pool. References to items in other packages are called *external*, and are represented in terms of tokens. References to items in the same CAP file are called *internal*, and are represented either in terms of tokens, or in a different internal format.

An external reference to a class is composed of a package token and a class token. Together those tokens specify a certain class in a certain package. An internal reference to a class is a 15-bit value which is a pointer to the class structure's location within the CAP file.

An external reference to a static class member, either a field or method, consists of a package token, a class token, and a token for the static field or static method. An internal reference to a static class member is a 16-bit value which is a pointer to the item's location in the CAP file.

References to instance fields, virtual methods and interface methods consist of a class reference and a token of the appropriate type. The class reference determines whether the reference is external or internal.

#### 4.1.5 Installation and Linking

External references in a CAP file can be resolved on a card from token form into the internal representation used by the card virtual machine.

A token can only be resolved in the context of the package which defines it. Just as the export file maps from a package's externally visible names to tokens, there is a set of link information for each package on the card that maps from tokens to resolved references.

---

### 4.2 Token Assignment

Tokens for an API are assigned by the API's developer and published in the package export file(s) for that API. Since the name-to-token mappings are published, an API developer may choose any order for tokens (subject to the constraints listed below).

A particular card platform can resolve tokens into whatever internal representation is most useful for that implementation of a Java Card VM. Some tokens may be resolved to indices. For example, an instance field token may be resolved to an index into a class instance's fields. In such cases, the token value is distinct from and unrelated to the value of the resolved index.

## 4.2.1 Token Details

Each kind of item in a package has its own independent scope for tokens of that kind. The token range and assignment rules for each kind are listed below.

TABLE 4-1 Token Range, Type and Scope

Token Type	Range	Type	Scope
Package	0 - 127	Private	CAP File
Class	0 - 255	Public	Package
Static Field	0 - 255	Public	Class
Static Method	0 - 255	Public	Class
Instance Field	0 - 255	Public or Private	Class
Virtual Method	0 - 127	Public or Private	Class Hierarchy
Interface Method	0 - 127	Public	Class

### 4.2.1.1 Package

All package references from within a CAP file are assigned private tokens; package tokens will never appear in an export file. Package token values must be in the range from 0 to 127, inclusive. The tokens for all the packages referenced from classes in a CAP file are numbered consecutively starting at zero. The ordering of package tokens is not specified.

### 4.2.1.2 Classes and Interfaces

All externally visible classes in a package are assigned public tokens. Package-visible classes are not assigned tokens. Class token values must be in the range from 0 to 255, inclusive. The tokens for all the public classes in a package are numbered consecutively starting at zero. The ordering of class tokens is not specified.

### 4.2.1.3 Static Fields

All externally visible static fields in a package are assigned public tokens. Package-visible and private static fields are not assigned tokens. No tokens are assigned for final static fields which are initialized to primitive, compile-time constants, as these fields are never linked on-card. Static fields token values must be in the range from 0 to 255, inclusive. The tokens for all other externally visible static fields in a class are numbered consecutively starting at zero. The ordering of static field tokens is not specified.

#### 4.2.1.4 Static Methods

All externally visible static methods in a package are assigned public tokens, including statically bound instance methods such as constructors. Static method token values must be in the range from 0 to 255, inclusive. Package-visible and private static methods are not assigned tokens. The tokens for all the externally visible static methods in a class are numbered consecutively starting at zero. The ordering of static method tokens is not specified.

#### 4.2.1.5 Instance Fields

All instance fields defined in a package are assigned either public or private tokens. Instance field token values must be in the range from 0 to 255, inclusive. Public and private tokens for instance fields are assigned from the same namespace. The tokens for all the instance fields in a class are numbered consecutively starting at zero, except that the token after an `int` field is skipped and the token for the following field is numbered two greater than the token of the `int` field. Tokens for externally visible fields must be numbered less than the tokens for package and private fields. For public tokens, the tokens for reference type fields must be numbered greater than the tokens for primitive type fields. For private tokens, the tokens for reference type fields must be numbered less than the tokens for primitive type fields. Beyond that the ordering of instance field tokens in a class is not specified.

public and protected = public tokens	primitive	boolean	0
		byte	1
		short	2
	references	byte[]	3
		Applet	4
package and private = private tokens	references	short[]	5
		Object	6
	primitive	int	7
		short	9

#### 4.2.1.6 Virtual Methods

All virtual methods defined in a package are assigned either public or private tokens. Virtual method token values must be in the range from 0 to 127, inclusive. Public and private tokens for virtual methods are assigned from different namespaces. The high bit of the byte containing a virtual method token is set to one if the token is a private token.

Public tokens for the externally visible introduced virtual methods in a class are numbered consecutively starting at one greater than the highest numbered public virtual method token of the class's superclass. If a method overrides a method implemented in the class's

superclass, that method uses the same token number as the method in the superclass. The high bit of the byte containing a public virtual method token is always set to zero, to indicate it is a public token. The ordering of public virtual method tokens in a class is not specified.

Private virtual method tokens are assigned differently from public virtual method tokens. If a class and its superclass are defined in the same package, the tokens for the package-visible introduced virtual methods in that class are numbered consecutively starting at one greater than the highest numbered private virtual method token of the class's superclass. If the class and its superclass are defined in different packages, the tokens for the package-visible introduced virtual methods in that class are numbered consecutively starting at zero. If a method overrides a method implemented in the class's superclass, that method uses the same token number as the method in the superclass. The definition of the Java programming language specifies that overriding a package-visible virtual method is only possible if both the class and its superclass are defined in the same package. The high bit of the byte containing a virtual method token is always set to one, to indicate it is a private token. The ordering of private virtual method tokens in a class is not specified.

#### 4.2.1.7 Interface Methods

All interface methods defined in a package are assigned public tokens, as interface methods are always public. Interface methods tokens values must be in the range from 0 to 127, inclusive. The tokens for all the interface methods defined in or inherited by an interface are numbered consecutively starting at zero. The token value for an interface method in a given interface is unrelated to the token values of that same method in any of the interface's superinterfaces. The high bit of the byte containing an interface method token is always set to zero, to indicate it is a public token. The ordering of interface method tokens is not specified.

## The Export File Format

---

This chapter describes the Java Card Virtual Machine **Export** file format. Compliant Java Card Virtual Machines must be capable of producing and consuming all **Export** files that conform to the specification provided in this book.

A Java Card **Export** file contains all of the public classes and interfaces defined in one Java package, and all of the public and protected fields and methods defined in those classes and interfaces. The **Export** File contains a mapping of these items to tokens (Chapter 3, “Java Card Naming”). Tokens are used to identify items in a **CAP** file (Chapter 5, “The Cap File Format”) and resolve references on a Java Card. The **Export** file is used during conversion of a package and creation of a Java Card **CAP** file.

The format in the **Export** file is similar to the Java **class** file. Structures and data in Java **class** files that describe internal implementation details, like bytecodes and private methods, are not included. Information beyond that in a Java **class** file includes package identification details and tokens associated with each class, interface, field, and method.

Final static fields of primitive types are represented in the **Export** file, but are not assigned token values suitable for referencing on Java Cards. These fields are not represented in **CAP** files or on Java Cards. Instead Java Card Converters must replace bytecodes that reference final static fields of primitive types with bytecodes that load the constant value of the field. Having this information included in the **Export** file enables this Converter functionality.<sup>1</sup>

An **Export** file consists of a stream of 8-bit bytes. All 16-bit quantities are constructed by reading in two consecutive 8-bit bytes. Multibyte data items are always stored in big-endian order, where the high-order bytes come first.

This chapter defines its own set of data types representing Java Card **Export** file data: The types **u1**, and **u2** represent an unsigned one-, and two-byte quantities, respectively.

1. Although Java compilers ordinarily replace references to final static fields of primitive types with primitive constants, this functionality is not required.

The Java Card Export file format is presented using pseudo structures written in a C-like structure notation. To avoid confusion with the fields of Java Card Virtual Machine classes and class instances, the contents of the structures describing the Java Card CAP file format are referred to as *items*. Unlike the fields of a C structure, successive items are stored in the Java Card file sequentially, without padding or alignment.

Variable-sized *tables*, consisting of variable-sized items, are used in several class file structures. Although we will use C-like array syntax to refer to table items, the fact that tables are streams of varying-sized structures means that it is not possible to directly translate a table index into a byte offset into the table.

A data structure that is referred to as an array, the elements are equal in size.

---

## 5.1 Export File Name

The name of a Export file must be the last portion of the package specification followed by the extension '.exp'. For example, the name of the Export file of the *javacard.framework* package must be *framework.exp*. Operating systems that impose limitations on file name lengths may transform an Export file's name according to its conventions.

CAP files are contained in a JAR file (Section 6.1.1, "Containment in a JAR File," on page 6-55). If an Export file is stored in a JAR file along with the CAP file, it must also be located in a directory called *javacard* that is a subdirectory package's directory. For example, the *framework.exp* file would be located in the subdirectory *javacard/framework/javacard*.

---

## 5.2 Export File

An Export file is defined by the following structure:

```
ExportFile {
    u4 magic
    u1 minor_version
    u1 major_version
    u2 constant_pool_count
    cp_info constant_pool[constant_pool_count]
    u2 this_package
    u2 export_class_count
    class_info classes[export_class_count]
}
```

The items in the ExportFile structure are as follows:

#### **magic**

The magic item contains the magic number identifying the ExportFile format; it has the value 0x00FACADE.

#### **minor\_version, major\_version**

The minor\_version and major\_version items are the minor and major version numbers of this Export file. An implementation of a Java Card Virtual Machine supports Export files having a given major version number and minor version numbers in the range 0 through some particular minor\_version.

If a Java Card Virtual Machine encounters a Export file with the supported major version but an unsupported minor version, the Java Card Virtual Machine must not attempt to interpret the content of the Export file. However, it will be feasible to upgrade a the Java Card Virtual Machine to support the newer minor version.

A Java Card Virtual Machine must not attempt to interpret a Export file with a different major version. A change of the major version number indicates a major incompatibility change, one that requires a fundamentally different Java Card Virtual Machine.

In this specification, the major version of the Export file has the value 2 and the minor version has the value 1. Only Sun Microsystems, Inc. may define the meaning and values of new Export file versions.

#### **constant\_pool\_count**

The constant\_pool\_count item is a non-zero, positive value that indicates the number of constants in the constant pool.

#### **constant\_pool[]**

The constant\_pool is a table of variable-length structures representing various string constants, class names, field names and other constants referred to within the ExportFile structure.

Each of the constant\_pool table entries, including entry zero, is a variable-length structure whose format is indicated by its first "tag" byte.

#### **this\_package**

The value of this\_package must be a valid index into the constant\_pool table. The constant\_pool entry at that index must be a CONSTANT\_Package\_info structure representing the package defined by this ExportFile.

#### **export\_class\_count**

The value of the export\_class\_count item gives the number of publicly accessi-

ble classes defined in this package.

classes[]

Each value of the `classes` table must be a variable-length `class_info` structure giving the complete description of a publicly accessible class declared in this package.

---

## 5.3 Constant Pool

All `constant_pool` table entries have the following general format:

```
cp_info {
    u1 tag
    u1 info[]
}
```

Each item in the `constant_pool` must begin with a 1-byte tag indicating the kind of `cp_info` entry. The content of the `info` array varies with the value of tag. The valid tags and their values are listed in TABLE 5-1. Each tag byte must be followed by two or more bytes giving information about the specific constant. The format of the additional information varies with the tag value.

TABLE 5-1 Constant Pool Tags

Constant Type	Value
CONSTANT_Package	13
CONSTANT_Interfacesref	7
CONSTANT_Integer	3
CONSTANT_Utf8	1

### 5.3.1 CONSTANT\_Package

The `CONSTANT_Package_info` structure is used to represent a package:

```
CONSTANT_Package_info {
    u1 tag
    u1 flags
    u2 name_index
    u1 minor_version
    u1 major_version
    u1 aid_length
    u1 aid[aid_length]
}
```

The items of the `CONSTANT_Package_info` structure are the following:

**tag**

The tag item has the value of `CONSTANT_Package` (13).

**flags**

The flags item is a mask of modifiers that apply to this package. The flags modifiers are shown in the following table.

TABLE 5-2 Package Flags

Flags	Value
ACC_LIBRARY	0x01

The `ACC_LIBRARY` flag has the value of one if this package does not define and declare any applets. In this case it is called a *library package*. Otherwise `ACC_LIBRARY` has the value of 0.

If the package is not a library package this export file may only contain shareable interfaces. A shareable interface is either the *javacard.framework.Sharable* interface or extends that interface.

---

**Note** – The restriction on exporting non-sharable items is imposed by the firewall defined in the *Java Card Runtime Environment 2.1* specifications.

---

All other flag values are reserved by the Java Card Virtual Machine. Their values must be zero.

**name\_index**

The value of the `name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing a valid Java package name.

As in Java class files, ASCII periods (‘.’) that normally separate the identifiers in a package name are replaced by ASCII forward slashes (‘/’). For example, the package name *javacard.framework* is represented in a `CONSTANT_Utf8_info` structure as *javacard/framework*.

**minor\_version, major\_version**

The `minor_version` and `major_version` items are the minor and major version numbers of this package. These values uniquely identify the particular implementation of this package and indicate the binary compatibility between packages. They are used to determine whether references between packages can be resolved, given the particular implementations.

A change in the minor version of a package represents an extension to the API and/or an update to the internal implementation that does not change the functionality of the API. It is guaranteed that two revisions of a package with the same major version and different minor versions have the same public token values assigned to the elements they have in common.

A change in the major version of a package represents an incompatibility with revisions of a lesser major version. This indicates that a portion of the API of the package has been deprecated and/or the implementation of the API is not functionally equivalent. Public tokens assigned to elements of the API that are defined in revisions of a package with different major versions are not required to have the same values.

A Java Card Virtual Machine may resolve references to an executable form of the package described in this Export file if the version associated with that executable has the same major version and a minor version that is less than or equal to this `minor_version` value.

The package provider is required to assign major and minor versions to different revisions of the package according to the constraints specified in this specification.

`aid_length`

The value of the `aid_length` item gives the number of bytes in the `aid` array. Valid values are between 5 and 16, inclusive.

`aid[]`

The `aid` array contains the ISO AID of the package. See ISO 7816-5 for a definition of an AID.

### 5.3.2 `CONSTANT_Interfaceref`

The `CONSTANT_Interfaceref_info` structure is used to represent an interface:

```
CONSTANT_Interfaceref_info {  
    u1 tag  
    u2 name_index  
}
```

The items of the `CONSTANT_Interfaceref_info` structure are the following:

`tag`

The `tag` item has the value of `CONSTANT_Interface (7)`.

name\_index

The value of the name\_index item must be a valid index into the constant\_pool table. The constant\_pool entry at that index must be a CONSTANT\_Utf8\_info structure representing a valid fully qualified Java interface name. These names are fully qualified because they may be defined in a package other than the one described in the Export File.

As in Java class files, ASCII periods (‘.’) that normally separate the identifiers in a class or interface name are replaced by ASCII forward slashes (‘/’). For example, the interface name sun.jc.interface1 is represented in a CONSTANT\_Utf8\_info structure as sun/jc/interface1.

### 5.3.3 CONSTANT\_Integer

The CONSTANT\_Integer\_info structure is used to represent four-byte numeric (int) constants:

```
CONSTANT_Integer_info {  
    u1 tag  
    u4 bytes  
}
```

The items of the CONSTANT\_Integer\_info structure are the following:

tag

The tag item has the value of CONSTANT\_Integer (3).

bytes

The bytes item of the CONSTANT\_Integer\_info structure contains the value of the *int* constant. The bytes of the value are stored in big-endian (high byte first) order.

### 5.3.4 CONSTANT\_Utf8

The CONSTANT\_Utf8\_info structure is used to represent constant string values. UTF-8 strings are encoded in the same way as described in *The Java Virtual Machine Specification*, Section 4.4.7.

The CONSTANT\_Utf8\_info structure is:

```

CONSTANT_Utf8_info {
    u1 tag
    u2 length
    u1 bytes[length]
}

```

The items of the `CONSTANT_Utf8_info` structure are the following:

**tag**

The tag item has the value of `CONSTANT_Utf8 (1)`.

**length**

The value of the `length` item gives the number of bytes in the bytes array (not the length of the resulting string). The strings in the `CONSTANT_Utf8_info` structure are not null-terminated.

**bytes[]**

The bytes array contains the bytes of the string. No byte may have the value (byte)0 or (byte)0xF0-(byte)0xFF.

---

## 5.4 Classes

Each interface and class is described by a variable-length `class_info` structure. The format of this structure is:

```

class_info {
    u1 token
    u2 access_flags
    u2 name_index
    u2 export_interfaces_count
    u2 interfaces[export_interfaces_count]
    u2 export_fields_count
    field_info fields[export_fields_count]
    u2 export_methods_count
    method_info methods[export_methods_count]
}

```

The items of the `class_info` structure are as follows:

**token**

The value of the `token` item is used to reference this class or interface on a Java Card. The tokens for all the public classes in a package are numbered consecutively starting at zero; beyond that the ordering of class tokens is not specified.

## access\_flags

The value of the `access_flags` item is a mask of modifiers used with class and interface declarations. The `access_flags` modifiers are shown in the following table.

TABLE 5-3 Class access and modifier flags

Name	Value	Meaning	Used By
ACC_PUBLIC	0x0001	Is public; may be accessed from outside its package	Class, interface
ACC_FINAL	0x0010	Is final; no subclasses allowed.	Class
ACC_INTERFACE	0x0200	Is an interface	Interface
ACC_ABSTRACT	0x0400	Is abstract; may not be instantiated	Class, interface
ACC_SHAREABLE	0x0800	Is shareable, may be shared between Java Card applets.	Class, interface

The `ACC_SHAREABLE` flag indicates whether this class or interface is shareable. A class is shareable if it implements (directly or indirectly) the `javacard.framework.shareable` interface. An interface is shareable if it is or implements (directly or indirectly) the `javacard.framework.Shareable` interface.

---

**Note** – The `ACC_SHAREABLE` flag is defined to enable Java Card Virtual Machines to implement the firewall restrictions defined by the *Java Card Runtime Environment 2.1* specifications.

---

All other class access and modifier flags are defined in the same way and with the same restrictions as described in *The Java Virtual Machine Specification*, Section 4.1.

The Java Card Virtual Machine reserves all other flag values. Their values must be zero.

## name\_index

The value of the `name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing a valid Java class name stored as a simple (not fully qualified) name, that is, as a Java identifier.<sup>1</sup>

1. In Java class files class names are fully qualified. In Java Card Export files all classes enumerated are defined in the package of the Export file making it unnecessary for class names to be fully qualified.

#### **export\_interfaces\_count**

The value of the `export_interface_count` item indicates the number of externally accessible interfaces implemented by this class or interface. It does not include package visible interfaces. It does include all superinterfaces in the hierarchy of public interfaces implemented by this class or interface.

#### **interfaces[]**

Each value in the `interfaces` array must be a valid index into the `constant_pool` table. The `constant_pool` entry at each value of `interfaces[i]`, where  $0 \leq i < \text{export\_interfaces\_count}$ , must be a `CONSTANT_Interfacesref_info` structure representing an interface which is an externally accessible superinterface of this class or interface type, in the left-to-right order given in the source for the type and its superclasses or superinterfaces.

#### **export\_fields\_count**

The value of the `export_fields_count` item gives the number of externally accessible fields declared by this class.

#### **fields[]**

Each value in the `fields` table must be a variable-length `field_info` structure giving a description complete enough to link to the field in a Java Card. The `field_info` structures represent all publicly accessible fields, both class variables and instance variables, declared by this class. It does not include items representing fields that are inherited from superclasses.

#### **export\_methods\_count**

The value of the `export_methods_count` item gives the number of externally accessible methods declared by this class.

#### **methods[]**

Each value in the `methods` table must be a `method_info` structure giving a description complete enough to link to the method in a Java Card. The `method_info` structures represent all publicly accessible class (static) methods implemented by this class, and all publicly accessible instance methods implemented by this class or its superclasses, or defined by this interface or its superinterfaces.

---

## 5.5 Fields

Each field is described by a variable-length `field_info` structure. The format of this structure is:

```
field_info {  
    u1 token  
    u2 access_flags  
    u2 name_index  
    u2 descriptor_index  
    u2 attributes_count  
    attribute_info attributes[attributes_count]  
}
```

The items of the `field_info` structure are as follows:

### token

The value of the `token` item is used to reference this field on a Java Card. There are three scopes for token values of fields: final static fields of primitives, other static fields, and instance fields.

The value of the `token` item for all final static fields of primitive types (*boolean*, *byte*, *short*, and *int*) is 0xFFFF. Final static fields of primitive types are represented in the `ExportFile` structure, but are not assigned token values suitable for on-card referencing. These fields are not represented in a CAP file or on Java Cards. Instead converters must replace bytecodes that reference final static fields of primitive types with bytecodes that load the constant value of the field.

The value of the `token` item for all other static fields in a class are numbered consecutively starting at zero, except that the token after an *int* field is skipped and the token for the following field is numbered two greater than the token of the *int* field. Beyond that the ordering of static field token assignment is not specified.

The value of the `token` item for all the instance fields in a class are numbered consecutively starting at zero, except that the token after an *int* field is skipped and the token for the following field is numbered two greater than the token of the *int* field. The tokens for non-reference fields must be numbered less than the tokens for reference fields; beyond that the ordering of instance fields is not specified.

### access\_flags

The value of the `access_flags` item is a mask of modifiers used with class and interface declarations. The `access_flags` modifiers are shown in the following table.

TABLE 5-4 Field access and modifier flags

Name	Value	Meaning	Used By
ACC_PUBLIC	0x0001	Is public; may be accessed from outside its package.	Any field
ACC_PROTECTED	0x0004	Is protected; may be accessed within subclasses.	Class field
ACC_STATIC	0x0008	Is static.	Class field
ACC_FINAL	0x0010	Is final; no further overriding or assignment after initialization.	Any field

Field access and modifier flags are defined in the same way and with the same restrictions as described in *The Java Virtual Machine Specification*, Section 4.5.

#### name\_index

The value of the name\_index item must be a valid index into the constant\_pool table. The constant\_pool entry at that index must be a CONSTANT\_Utf8\_info structure representing a valid Java field name stored as a simple (not fully qualified) name, that is, as a Java identifier.

#### descriptor\_index

The value of the descriptor\_index item must be a valid index into the constant\_pool table. The constant\_pool entry at that index must be a CONSTANT\_Utf8\_info structure representing a valid Java field descriptor.

Representation of a field descriptor in an Export File is the same as in a Java Class file. See the specification described in *The Java Virtual Machine Specification*, Section 4.3.2.

#### attributes\_count

The value of the attributes\_count item indicates the number of additional attributes of this field. For static final fields the value must be 1; that is, when both the ACC\_STATIC and ACC\_FINAL bits in the flags item are set an attribute must be present. For all other fields the value must be 0.

#### attributes[]

The only attribute defined for the attributes table of a field\_info structure by this specification is the ConstantValue attribute. This must be defined for static final fields of primitives (*boolean*, *byte*, *short*, and *int*).

---

## 5.6 Methods

Each method is described by a variable-length `method_info` structure. The format of this structure is:

```
method_info {  
    u1 token  
    u2 access_flags  
    u2 name_index  
    u2 descriptor_index  
}
```

The items of the `method_info` structure are as follows:

### `token`

The value of the `token` item is used to reference this method on a Java Card. There are two scopes for token values of methods: static methods, and instance methods.

The static method scope includes constructors. The tokens for all the static methods and constructors in a class are numbered consecutively starting at zero. Beyond that ordering is not specified.

The tokens for introduced instance methods in a class are numbered consecutively starting at one greater than the highest numbered instance method token of the class's superclass. If a method overrides a method implemented in the class's superclass, that method uses the same token number as the method in the superclass. Beyond that the ordering of instance methods is not specified.

The maximum token value for an instance method is 127.

### `access_flags`

The value of the `access_flags` item is a mask of modifiers used with class and interface declarations. The `access_flags` modifiers are shown in the following table.

TABLE 5-5 Method access and modifier flags

Name	Value	Meaning	Used By
ACC_PUBLIC	0x0001	Is public; may be accessed from outside its package.	Any method
ACC_PROTECTED	0x0004	Is protected; may be accessed within subclasses.	Class/instance method
ACC_STATIC	0x0008	Is static.	Class/instance method
ACC_FINAL	0x0010	Is final; no further overriding or assignment after initialization.	Class/instance method
ACC_ABSTRACT	0x0400	Is abstract; no implementation is provided	Any method

Method access and modifier flags are defined in the same way and with the same restrictions as described in *The Java Virtual Machine Specification, Section 4.6*.

Unlike in Java class files, the ACC\_NATIVE flag is not supported in Export files. Whether a method is native is an implementation detail that is not relevant to referencing packages.

#### name\_index

The value of the name\_index item must be a valid index into the constant\_pool table. The constant\_pool entry at that index must be a CONSTANT\_Utf8\_info structure representing either the special internal method name for constructors, <init>, or a valid Java method name stored as a simple (not fully qualified) name.

#### descriptor\_index

The value of the descriptor\_index item must be a valid index into the constant\_pool table. The constant\_pool entry at that index must be a CONSTANT\_Utf8\_info structure representing a valid Java method descriptor.

Representation of a method descriptor in an Export File is the same as in a Java Class file. See the specification described in *The Java Virtual Machine Specification, Section 4.3.3*.

---

## 5.7 Attributes

Attributes are used in the field\_info structure of the Export File format. All attributes have the following general format:

```

attribute_info {
    u2 attribute_name_index
    u4 attribute_length
    u1 info[attribute_length]
}

```

### 5.7.1 ConstantValue Attribute

The `ConstantValue` attribute is a fixed-length attribute used in the `attributes` table of the `field_info` structures. A `ConstantValue` attribute represents the value of a constant field that must be final static; that is, both the `ACC_STATIC` and `ACC_FINAL` bits in the `flags` item of the `field_info` structure must be set. There can be no more than one `ConstantValue` attribute in the `attributes` table of a given `field_info` structure. The constant field represented by the `field_info` structure is assigned the value referenced by its `ConstantValue` attribute as part of its initialization.

The `ConstantValue` attribute has the format

```

ConstantValue_attribute {
    u2 attribute_name_index
    u4 attribute_length
    u2 constantvalue_index
}

```

The items of the `ConstantValue_attribute` structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the string “ConstantValue”.

`attribute_length`

The value of the `attribute_length` item of a `ConstantValue_attribute` structure must be 2.

`constantvalue_index`

The value of the `constantvalue_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must give the constant value represented by this attribute.

The `constant_pool` entry must be of a type `CONSTANT_Integer`.

**This Page Blank (uspto)**

## The Cap File Format

---

This chapter describes the Java Card Virtual Machine CAP (Converted Applet) file format. Each CAP file contains all of the classes and interfaces defined in one Java package. Compliant Java Card Virtual Machines must be capable of producing and consuming all CAP files that conform to the specification provided in this book.

A CAP file consists of a stream of 8-bit bytes. All 16-bit and 32-bit quantities are constructed by reading in two and four consecutive 8-bit bytes, respectively. Multibyte data items are always stored in big-endian order, where the high-order bytes come first.

This chapter defines its own set of data types representing Java Card CAP file data: The types `u1`, and `u2` represent an unsigned one-, and two-byte quantities, respectively. Some `u1` types are represented as *bitfield* structures, consisting of arrays of bits. The zeroeth bit in each bit array represents the most significant bit.

The Java Card CAP file format is presented using pseudo structures written in a C-like structure notation. To avoid confusion with the fields of Java Card Virtual Machine classes and class instances, the contents of the structures describing the Java Card CAP file format are referred to as *items*. Unlike the fields of a C structure, successive items are stored in the Java Card file sequentially, without padding or alignment.

Variable-sized *tables*, consisting of variable-sized items, are used in several class file structures. Although we will use C-like array syntax to refer to table items, the fact that tables are streams of varying-sized structures means that it is not possible to directly translate a table index into a byte offset into the table.

A data structure that is referred to as an array, the elements are equal in size.

---

**Note** – Many elements in this CAP file specifications are assigned token values. The definitions of these tokens are given in Chapter 4, “Java Card Naming.”

---

## 6.1 Component Model

A CAP file is represented as a set of components. A CAP file is not complete unless all of the components specified in this chapter are present, except the Applet and Export Components. The Applet Component ("Applet Component" on page 61) is required if the package defines one or more Java Card applets. The Export Component ("Export Component" on page 85) is required if the package is to be made available to other packages on a Java Card. All components have the following general format:

```
component {  
    u1 tag  
    u2 size  
    u1 info[]  
}
```

Each component begins with a 1-byte tag indicating the kind of component. Valid tags and their values are listed in the next table. The size item indicates the number of bytes in the info array of the component, not including the tag and size items. The content and format of the info array varies with the type of component.

TABLE 6-1 Component Tags

Component Type	Value
COMPONENT_Header	1
COMPONENT_Directory	2
COMPONENT_Applet	3
COMPONENT_Imports	4
COMPONENT_ConstantPool	5
COMPONENT_Class	6
COMPONENT_Method	7
COMPONENT_StaticField	8
COMPONENT_ReferenceLocation	9
COMPONENT_Export	10
COMPONENT_Descriptor	11

Sun may define additional components in future versions of the CAP file specification.

## 6.1.1 Containment in a JAR File

All CAP file components are represented in individual files stored in a JAR File. The JAR path/file names are enumerated in the following table. The names of each of the component files are not case sensitive. Operating systems that impose limitations on file name lengths may transform an component file names according to its conventions.

TABLE 6-2 JAR File Names

Component Type	File Name
COMPONENT_Header	Header.cap
COMPONENT_Directory	Directory.cap
COMPONENT_Applet	Applet.cap
COMPONENT_Imports	Imports.cap
COMPONENT_ConstantPool	ConstantPool.cap
COMPONENT_Class	Class.cap
COMPONENT_Method	Method.cap
COMPONENT_StaticField	StaticField.cap
COMPONENT_ReferenceLocation	RefLocations.cap
COMPONENT_Export	Export.cap
COMPONENT_Descriptor	Descriptor.cap

The location within a JAR file of the files that constitute a CAP file is in a directory called *javacard* that is a subdirectory representing the package's directory. For example, the CAP file component files for the package *javacard.framework* are located in the subdirectory *javacard/framework/javacard*.

The name of a JAR file containing a CAP file is not defined as part of this specification. Other files, including other CAP files, may also reside in the JAR file.

## 6.1.2 Defining New Components

Java Card CAP files are permitted to contain new components. All components not defined as part of this Java Card Virtual Machine specification must not affect the semantics of the content of the specified components, and Java Card Virtual Machines must be able to accept CAP files that do not contain new components. Java Card Virtual Machine implementations are required to ignore components they do not recognize.

New components are named in two ways. They are assigned both an ISO 7816-5 AID and a tag. Valid tag values are between 128 and 255, inclusive. The Directory Component ("Directory Component" on page 59) must contain an entry in the `custom_component` item specifying both the AID and associated tag. The new component must contain the tag value as its first item.

The JAR path/file names of custom components must follow the same restrictions as those specified in Section 6.1.1, "Containment in a JAR File." That is, they must be located in the `<package_directory>/javacard` subdirectory of the JAR file and must have the extension `'cap'`.

---

## 6.2 Header Component

The Header Component is described by the following variable-length structure:

```
header_component {  
    u1 tag  
    u2 size  
    u4 magic  
    u1 minor_version  
    u1 major_version  
    u1 flags  
    package_info this_package  
}
```

The items in the `header_component` structure are as follows:

### tag

The tag item has the value `COMPONENT_Header(1)`.

### size

The size item indicates the number of bytes in the `header_component` structure, excluding the tag and size items.

### magic

The magic item supplies the magic number identifying the Java Card CAP file format; it has the value `0xDECAFFED`.

### minor\_version, major\_version

The `minor_version` and `major_version` items are the minor and major version numbers of this CAP file. An implementation of a Java Card Virtual Machine must support CAP files having a specific major version number and minor version numbers in the range of 0 through some particular `minor_version`.

If a Java Card Virtual Machine encounters a CAP file with the supported major version but an unsupported minor version, the Java Card Virtual Machine must not attempt to interpret the content of the CAP file. However, it will be feasible to upgrade a the Java Card Virtual Machine to support the newer minor version.

A Java Card Virtual Machine must not attempt to interpret a CAP file with a different major version. A change of the major version number indicates a major incompatibility change, one that requires a fundamentally different Java Card Virtual Machine.

In this specification, the major version of the CAP file has the value 2 and the minor version has the value 1. Only Sun Microsystems, Inc. may define the meaning and values of new CAP file versions.

## flags

The flags item is a mask of modifiers that apply to this package. The flags modifiers are shown in the following table.

TABLE 6-3 Package Flags

Flags	Value
ACC_INT	0x01
ACC_EXPORT	0x02
ACC_APPLET	0x04

The ACC\_INT flag has the value of one if the Java *int* type is present in this package. This includes fields defined as type *int*, fields defined as type *int* array, or instructions of type *int*. Otherwise the ACC\_INT flag has the value of 0.

The ACC\_EXPORT flag has the value of one if other packages may link to this package. Otherwise it has the value of 0. If the ACC\_EXPORT flag has the value of one an Export Component ("Export Component" on page 85) must be defined in this CAP file. If the ACC\_EXPORT flag has the value of zero an Export Component must not be defined in the CAP file.

The ACC\_APPLET flag has the value of one if one or more Java Card applets are defined this package. Otherwise it has the value of 0. If the ACC\_APPLET flag has the value of one an Applet Component ("Applet Component" on page 61) must be defined in this CAP file. If the ACC\_APPLET flag has the value of zero an Applet Component must not be defined in the CAP file.

The Java Card Virtual Machine reserves all other flag values. Their values must be zero.

## **this\_package**

The **this\_package** item describes the package defined in this CAP file. It is represented as a **package\_info** structure:

```
package_info {  
    u1 minor_version  
    u1 major_version  
    u1 AID_length  
    u1 AID[AID_length]  
}
```

## **minor\_version, major\_version**

The **minor\_version** and **major\_version** items are the minor and major version numbers of this package. These values uniquely identify the particular implementation of this package and indicate the binary compatibility between packages. These versions are used when resolving references between packages in preparation for execution.

A change in the minor version of a package represents an extension to the API and/or an update to the internal implementation that does not change the functionality of the API. It is guaranteed that two revisions of a package with the same major version and different minor versions have the same public token values assigned to the elements they have in common.

A change in the major version of a package represents an incompatibility with revisions of a lesser major version. This indicates that a portion of the API of the package has been deprecated and/or the implementation of the API is not functionally equivalent. Public tokens assigned to elements of the API that are defined in revisions of a package with different major versions are not required to have the same values.

A Java Card Virtual Machine may link another package to this package if and only if the other package's CAP file has been constructed using a revision of this package that has the same major version and a minor version less than or equal to the minor version specified by the **major\_version** and **minor\_version** items. The Imports Component Section 6.5, "Imports Component," on page 6-62 in the CAP file of the referencing package contains both the Java Card name (AID) of the referenced package and its major and minor versions.

The package provider is required to assign major and minor versions to different revisions of the package according to the constraints specified in this specification.

## **AID\_length**

The AID\_length item represents the length of the AID item. Valid values are between 5 and 16, inclusive.

#### AID[]

The AID item represents the Java Card name of the package. See ISO 7816-5 for the definition of an AID.

---

## 6.3 Directory Component

The Directory Component lists the size of each of the components defined in this CAP file. It is described by the following variable-length structure:

```
directory_component {
    u1 tag
    u2 size
    u1 basic_count
    u1 custom_count
    { u1 component_tag
      u2 size
    } basic_components[basic_count]
    { u1 component_tag
      u2 size
      u1 AID_length
      u1 AID[AID_length]
    } custom_components[custom_count]
}
```

The items in the directory\_component structure are as follows:

#### tag

The tag item has the value COMPONENT\_Directory(2).

#### size

The size item indicates the number of bytes in the directory\_component structure, excluding the tag and size items.

#### basic\_count

The basic\_count item indicates the number of basic components defined in this CAP file. These components are defined in this specification. Valid values for the basic\_count item are 9, 10 and 11, depending upon whether an Export Component and/or Applet Component are defined.

#### **custom\_count**

The **custom\_count** item indicates the number of components defined in this CAP file that are not defined in this specification. Valid values are between 0 and 127, inclusive.

#### **basic\_components[]**

The **basic\_components** item represents an array of structures, each representing a component defined in this CAP file. The items in the structure are:

##### **component\_tag**

The **component\_tag** item represents the tag of the component. Valid values are defined in Table 6-1, "Component Tags," on page 54.

##### **size**

Except in the case of the Static Field Component, the **size** item represents the size in bytes in the component, including the tag and size items.

For the Static Field Component the **size** item represents the size in bytes of the Static Field Image. This value is equal to the **byte\_count** item in the **static\_field\_component** structure. It does not include the number of bytes required to store the array instances defined in the Static Field Component.

#### **custom\_components[]**

The **custom\_components** item represents an array of variable-length structures that name each of the components in this CAP file that are not defined by this standard. Each component is assigned an AID conforming to the ISO 7816-5 standard. The RID (first 5 bytes) of all of the custom component's AID must have the same value. In addition, the RID of the custom component's AID must have the same value as the RID of the package defined in this CAP file.

The items in entries of the **custom\_component** table are:

##### **component\_tag**

The **component\_tag** item represents the tag of the component. Valid values are between 128 and 255, inclusive.

##### **size**

The **size** item represents the size in bytes of the component.

##### **AID\_length**

The **AID\_length** item represents the length of the AID item. Valid values are between 5 and 16, inclusive.

AID[]

The AID item represents the Java Card name of the component. See ISO 7816-5 for the definition of an AID.

---

## 6.4 Applet Component

The Applet Component describes each of the applets defined in this package. It is described by the following variable-length structure

```
applet_component {  
    u1 tag  
    u2 size  
    u1 count  
    { u1 AID_length  
      u1 AID[AID_length]  
      u2 install_method_offset  
    } applets[count]  
}
```

The items in the `applet_component` structure are as follows:

**tag**

The tag item has the value `COMPONENT_Applet(3)`.

**size**

The size item indicates the number of bytes in the `applet_component` structure, excluding the tag and size items.

**count**

The count item indicates the number of applets defined in this package.

**applets[]**

The `applets` item represents an array of structures that describe each of the applets defined in this package. Each applet is assigned an AID conforming to the ISO 7816-5 standard. The RID (first 5 bytes) of all of the applet's AID must have the same value. In addition, the RID of each applet's AID must have the same value as the RID of the package defined in this CAP file.

The items in the `applets` structure are defined as follows:

**AID\_length**

The `AID_length` item represents the length of the AID item. Valid values

are between 5 and 16, inclusive.

#### AID[]

The AID item represents the Java Card name of the applet.

#### install\_method\_offset

The value of the `install_method_offset` item must be a 16-bit offset into the `info` item of the Method Component. The item at that offset must be a `method_info` structure that represents the static `install()` method of a class that extends *javacard.framework.applet*. This method is called to initialize the applet.

- Restrictions placed on the *install()* method of an applet are imposed by the *Java Card 2.1 Runtime Environment Specification*.

---

## 6.5 Imports Component

The Imports Component enumerates the packages that are referenced by this package. It does not include the package defined in this CAP file. The Imports Component is represented by the following structure:

```
imports_component {  
    u1 tag  
    u2 size  
    u1 count  
    package_info packages[count]  
}
```

The items in the `imports_component` structure are as follows:

#### tag

The tag item has the value `COMPONENT_Imports(4)`.

#### size

The `size` item indicates the number of bytes in the `imports_component` structure, excluding the tag and size items.

#### count

The `count` item indicates the number of items in the `packages` table. The value of the `count` item must be less than or equal to 127.

`packages[]`

The `packages` item represents a table of variable-length `package_info` structures as defined for “`this_package`” on page 58. The table contains an entry for each of the packages referenced in the CAP file, not including the package defined.

The major and minor versions specified in the `package_info` structure are equal to the major and minor versions specified in the referenced package’s Export File. Java Card Virtual Machines must compare these values with the version associated with the executable image of the referenced package in order to determine whether the two are compatible. Compatibility rules are specified in “`minor_version`, `major_version`” on page 58.

The value of a `package_token` assigned to an imported package must be equal to the index into the `packages` table to the entry for that package.

---

## 6.6 Constant Pool Component

The Constant Pool Component describes the classes, methods, and fields referenced in the Method Component (“Method Component” on page 76) of this package. The component is described by the following structure:

```
constant_pool_component {  
    u1 tag  
    u2 size  
    u2 count  
    cp_info constant_pool[count]  
}
```

The items in the `constant_pool_component` structure are as follows:

**tag**

The tag item has the value `COMPONENT_ConstantPool(5)`.

**size**

The size item indicates the number of bytes in the `constant_pool_component` structure, excluding the tag and size items.

**count**

The count item represents the number entries in the constants array.

## constant\_pool

The constant\_pool item represents a table of cp\_info structures:

```
cp_info {  
    u1 tag  
    u1 info[]  
}
```

Each item in the constant\_pool table represents a 4-byte structure. Each structure must begin with a 1-byte tag indicating the kind of cp\_info entry. The content of the info array varies with the value of the tag. The valid tags and their values are listed in the following table. The format of the additional information varies with the tag value.

TABLE 6-4 Constant Pool Tags

Constant Type	Tag
CONSTANT_Classref	1
CONSTANT_InstanceFieldref	2
CONSTANT_VirtualMethodref	3
CONSTANT_SuperMethodref	4
CONSTANT_StaticFieldref	5
CONSTANT_StaticMethodref	6

Java Card constant types are more specific than those in Java class files. The categories are based not only on the type of the item references but also on the manner in which it is referenced and located in other CAP file components. For example virtual methods invoked using the Java *super* keyword are located through the virtual method tables in a class\_info structure (Section 6.7.1, “interface\_info and class\_info,” on page 6-71), but unlike virtual method references can be resolved to direct references on a Java Card.

There are no ordering constraints on constant pool entries. It is recommended, however, that CONSTANT\_InstanceFieldref constants occur first.

### 6.6.1 CONSTANT\_Classref

The CONSTANT\_Classref\_info structure is used to represent a reference to a class or an interface:

```
CONSTANT_Classref_info {  
    u1 tag  
    union {  
        u2 internal_class_ref  
        { u1 package_token  
          u1 class_token
```

```

        } external_class_ref
    } class_ref
    u1 padding
}

```

The items in the `CONSTANT_Classref_info` structure are the following:

**tag**

The tag item has the value `CONSTANT_Classref(1)`.

**class\_ref**

The `class_ref` item represents a reference to a class or interface. If the class or interface is defined in this package the structure represents an `internal_class_ref` and the high bit of the structure is zero. If the class or interface is defined in another package the structure represents an `external_class_ref` and the high bit of the structure is one.

The value of the `class_ref` item must not be `0xFFFF`. This value is reserved to indicate a null class reference in the `super_class_ref` item of `class_info` structures.

**internal\_class\_ref**

The `internal_class_ref` structure represents a 16-bit offset into the `info` item of the Class Component to the `interface_info` or `class_info` structure of this class or interface. This value must be less than or equal to 32767, inclusive, making the high bit equal to zero.

**external\_class\_ref**

The `external_class_ref` structure represents a reference to a class or interface defined in another package. The high bit of this structure is one.

**package\_token**

The `package_token` item represents an 8-bit package token scoped to the CAP file. The value of this token must be equal to the index into the set of packages listed in the Imports Component to the definition of the reference package. The value of the `package_token` item must be less than or equal to 127.

The high bit of the `package_token` item must be one.

**class\_token**

The `class_token` item represents the token of the class or interface in the referenced package.

padding

The padding item has the value zero. It is present to make the size of a `CONSTANT_Classref_info` structure the same as all other constants in the `constant_pool`.

## 6.6.2 `CONSTANT_InstanceFieldref`, `CONSTANT_VirtualMethodref`, and `CONSTANT_SuperMethodref`

References to instance fields, and virtual methods are represented by similar structures:

```
CONSTANT_InstanceFieldref_info {  
    u1 tag  
    class_ref class  
    u1 instance_field_token  
}  
  
CONSTANT_VirtualMethodref_info {  
    u1 tag  
    class_ref class  
    u1 virtual_method_token  
}  
  
CONSTANT_SuperMethodref_info {  
    u1 tag  
    class_ref class  
    u1 virtual_method_token  
}
```

The items in these structures are as follows:

tag

The tag item of a `CONSTANT_InstanceFieldref_info` structure has the value `CONSTANT_InstanceFieldref(2)`.

The tag item of a `CONSTANT_VirtualMethodref_info` structure has the value `CONSTANT_VirtualMethodref(3)`.

The tag item of a `CONSTANT_SuperMethodref_info` structure has the value `CONSTANT_SuperMethodref(4)`.

## class

The **class** item represents a **class\_ref** structure (Section 6.6.1, “**CONSTANT\_Classref**,” on page 6-64). If the referenced class is defined in another package the high bit of this structure has the value one. If the referenced class is defined in this package the high bit of this structure has the value of zero.

The class referenced in the **CONSTANT\_InstanceField\_info** structure must represent the class that contains a declaration of the field.

The class referenced in the **CONSTANT\_VirtualMethodref\_info** structure must represent a class that contains a declaration or definition of the method.

The class referenced in the **CONSTANT\_SuperMethodref\_info** structure must represent the class that contains the super invocation.

## token

The token item in the **CONSTANT\_InstanceFieldref\_info** structure represents an instance field token, within the scope the referenced class.

The token item of the **CONSTANT\_VirtualMethodref\_info** structure represents a virtual method token within the scope of the referenced class. If the token item represents a public or protected virtual method the high bit is zero. If the token item represents a package-visible virtual method the high bit is one. In this case the **class** item must represent a reference to a class defined in this package.

The token item of the **CONSTANT\_SuperMethodref\_info** structure represents a virtual method token. Unlike in the **CONSTANT\_VirtualMethodref\_info** structure, this item represents the token of the virtual method in the scope of the superclass of the class indicated by the **class** item. If the token item represents a public or protected virtual method the high bit is zero. If the token item represents a package-visible virtual method the high bit is one. In this case the **class** item must represent a reference to a class defined in this package and the immediate superclass of the **class** item must also be defined in this package.

### 6.6.3 CONSTANT\_StaticFieldref and CONSTANT\_StaticMethodref

References to static fields and methods are represented by similar structures:

```
CONSTANT_StaticFieldref_info {
    u1 tag
    union {
        { u1 padding
          u2 offset
        } internal_ref
        { u1 package_token
          u1 class_token
          u1 token
        } external_ref
    } static_field_ref
}

CONSTANT_StaticMethodref_info {
    u1 tag
    union {
        { u1 padding
          u2 offset
        } internal_ref
        { u1 package_token
          u1 class_token
          u1 token
        } external_ref
    } static_method_ref
}
```

The items in these structures are as follows:

#### tag

The tag item of a CONSTANT\_StaticFieldref\_info structure has the value CONSTANT\_StaticFieldref(6).

The tag item of a CONSTANT\_StaticMethodref\_info structure has the value CONSTANT\_StaticMethodref(7).

#### static\_field\_ref and static\_method\_ref

The static\_field\_ref and static\_method\_ref item represents a reference to a static field or static method, respectively. Static method references include references to static methods, constructors, and private instance methods.

If the referenced item is defined in this package the structure represents an

internal\_ref and the high bit of the structure is zero. If the referenced item is defined in another package the structure represents an external\_ref and the high bit of the structure is one.

#### internal\_ref

The internal\_ref item represents a reference to a static or method defined in this package. The items in the structure are:

##### padding

The padding item is equal to 0.

##### offset

The offset item of a CONSTANT\_StaticFieldref\_info structure represents a 16-bit offset into the Static Field Image to this static field.

The offset item of a CONSTANT\_StaticMethodref\_info structure represents a 16-bit offset into the info item of the Method Component to a method\_info structure. The method\_info structure must represent the referenced method.

#### external\_ref

The external\_ref item represents a reference to a static field or method defined in another package. The items in the structure are:

##### package\_token

The package\_token item represents an 8-bit package token scoped to the CAP file. The value of this token must be equal to the index into the set of packages listed in the Imports Component to the definition of the referenced package. The value of the package\_token item must be less than or equal to 127.

The high bit of the package\_token item must be one.

##### class\_token

The class\_token item represents the token of the class in the referenced package.

##### token

The token item of a CONSTANT\_StaticFieldref\_info structure represents a static field token. It is the token of the static field in the referenced package.

The token item of a CONSTANT\_StaticMethodref\_info structure represents a static method token. It is the token of the static method in the referenced package.

---

## 6.7 Class Component

The Class Component describes each of the classes and interfaces defined in this package. The information included for each interface is sufficient to uniquely identify the interface and to test whether or not a cast is valid.

The information included for each class is sufficient to resolve operations associated with instances of a class. The operations include creating an instance, testing whether or not a cast is valid, dispatching virtual method invocations, and dispatching interface method invocations. Also included is sufficient information to locate instance fields of type reference.

The Class Component is represented by the following structure:

```
class_component {  
    u1 tag  
    u2 size  
    interface_info interfaces[]  
    class_info classes[]  
}
```

The items in the `class_component` structure are as follows:

**tag**

The tag item has the value `COMPONENT_Class(6)`.

**size**

The size item indicates the number of bytes in the `class_component` structure, excluding the tag and size items.

**interfaces[]**

The `interfaces` item represents an array of variable-length `interface_info` structures. Each interface defined in this package is represented in the array. The entries are ordered based on hierarchy such that a superinterface has a lower index than any of its subinterfaces.

**classes[]**

The `classes` item represents an array of variable-length `class_info` structures. Each class defined in this package is represented in the array. The entries are ordered based on hierarchy such that a superclass has a lower index than any of its subclasses.

## 6.7.1 interface\_info and class\_info

The `interface_info` and `class_info` structures represent interfaces and classes, respectively. The `interface_info` structure is a strict subset of the `class_info` structure. The two are differentiated by the value of the first bit in the structure. The structures are defined as follows:

```
interface_info {
    u1 bitfield {
        bit[4] flags
        bit[4] reserved
    }
}

class_info {
    u1 bitfield {
        bit[4] flags
        bit[4] interface_count
    }
    class_ref super_class_ref
    u1 declared_instance_size
    u1 first_reference_index
    u1 reference_count
    u1 public_method_table_base
    u1 public_method_table_count
    u1 package_method_table_base
    u1 package_method_table_count
    u2 public_virtual_method_table[public_method_table_count]
    u2 package_virtual_method_table[package_method_table_count]
    implemented_interface_info interfaces[interface_count]
}
```

The items of the `interface_info` and `class_info` structure are as follows:

### flags[]

The `flags` item is a mask of modifiers used to describe this interface or class. Valid values are shown in the following table:

TABLE 6-5 Interface and Class Info Flags

Name	Value
ACC_INTERFACE	0x8
ACC_SHAREABLE	0x4

The `ACC_INTERFACE` flag indicates whether this `interface_info` or `class_info` structure represents an interface or a class. The value must be 1 if it represents an `interface_info` structure and 0 if a `class_info` structure.

The `ACC_SHAREABLE` flag in an `interface_info` structure indicates whether this interface is shareable. The value of this flag must be one if and only if the interface

is *javacard.framework.Shareable* interface or implements that interface directly or indirectly.

The ACC\_SHAREABLE flag in an *class\_info* structure indicates whether this class is shareable. The value of this flag must be one if and only if this class or any of its superclasses implements an interface that is shareable.

---

**Note** – A Java Card Virtual Machine uses the ACC\_SHAREABLE flag to implement the firewall restrictions defined by the *Java Card Runtime Environment 2.1* specifications.

---

The Java Card Virtual Machine reserves all other flag values. Their values must be zero.

reserved

The reserved item of the *interface\_info* structure has the value of zero.

*interface\_count*[]

The *interface\_count* item of the *class\_info* structure indicates the number of entries in the *interfaces* table item.

*super\_class\_ref*

The *super\_class\_ref* item of the *class\_info* structure is a *class\_ref* structure representing the superclass of this class. The *class\_ref* structure is defined as part of the *CONSTANT\_Classref\_info* structure (Section 6.6.1, “CONSTANT\_Classref,” on page 6-64).

If this class does not have a superclass, the value of the *super\_class\_ref* item must be 0xFFFF.

*declared\_instance\_size*

The *declared\_instance\_size* item of the *class\_info* structure represents the number of 16-bit cells required to represent the instance fields declared by this class. It does not include instance fields declared by superclasses of this class.

Instance fields of type *int* are represented in two 16-bit cells, while all other field types are represented in one 16-bit cell.

*first\_reference\_token*

The *first\_reference\_token* item of the *class\_info* structure represents the token value of the first reference-type instance field defined by this class. It does not include instance fields defined by superclasses of this class.

#### **reference\_count**

The **reference\_count** item of the **class\_info** structure represents the number of reference-type instance field defined by this class. It does not include reference-type instance fields defined by superclasses of this class.

#### **public\_method\_table\_base**

The **public\_method\_table\_base** item of the **class\_info** structure must be equal to the token value of the first method in the **public\_virtual\_method\_table** array.

#### **public\_method\_table\_count**

The **public\_method\_table\_count** item of the **class\_info** structure indicates the number of entries in the **public\_virtual\_method\_table** array.

If this class does not define any public or protected override methods, the minimum valid value of **public\_method\_table\_count** item is the number of public and protected virtual methods declared by this class. If this class defines one or more public or protected override methods, the minimum valid value of **public\_method\_table\_count** item is the value of the largest public or protected virtual method token, minus the value of the smallest public or protected virtual method token, plus one.

The maximum valid value of the **public\_method\_table\_count** item is the value of the largest public or protected virtual method token, plus one.

Any value for the **public\_method\_table\_count** item between the minimum and maximum specified here are valid. However, the value must correspond to the number of entries in the **public\_virtual\_method\_table**.

#### **package\_method\_table\_base**

The **package\_method\_table\_base** item of the **class\_info** structure must be equal to the token value of the first entry in the **package\_virtual\_method\_table** array.

#### **package\_method\_table\_count**

The **package\_method\_table\_count** item of the **class\_info** structure indicates the number of entries in the **package\_virtual\_method\_table** array.

If this class does not define any override methods, the minimum valid value of **package\_method\_table\_count** item is the number of package visible virtual methods declared by this class. If this class defines one or more package visible override methods, the minimum valid value of **package\_method\_table\_count** item is the value of the largest package visible virtual method token, minus the value of the smallest package visible virtual method token, plus one.

The maximum valid value of the **package\_method\_table\_count** item is the

value of the largest package visible method token, plus one.

Any value for the `package_method_table_count` item between the minimum and maximum specified here are valid. However, the value must correspond to the number of entries in the `package_virtual_method_table`.

#### `public_virtual_method_table`

The `public_virtual_method_table` item of the `class_info` structure represents an array of public and protected virtual methods that can be invoked on an instance of this class. It includes methods declared or defined by this class. It may also include methods declared or defined by any or all of its superclasses. The value of an index into this table must be equal to the value of the virtual method token of the indicated method minus the value of the `public_method_table_base` item.

Entries in the `public_virtual_method_table` array that represent methods defined or declared in this package contain offsets into the `info` item of the Method Component (Section 6.8, "Method Component," on page 6-76) to the `method_info` structure representing the method. Entries that represent methods defined or declared in another package contain the value `0xFFFF`.

Entries for methods that are declared abstract, not including those defined by interfaces, are represented in the `public_virtual_method_table` array in the same way as non-abstract methods.

#### `package_virtual_method_table`

The `package_virtual_method_table` item of the `class_info` structure represents an array of package virtual methods that can be invoked on an instance of this class. It includes methods declared or defined by this class. It may also include methods declared or defined by any or all of its superclasses that are defined in this package. The value of an index into this table must be equal to the value of the virtual method token of the indicated method & `0x7F`, minus the value of the `package_method_table_base` item.

All entries in the `package_virtual_method_table` array represent methods defined or declared in this package. They contain offsets into the `info` item of the Method Component ("Method Component" on page 76) to the `method_info` structure representing the method.

Entries for methods that are declared abstract, not including those defined by interfaces, are represented in the `package_virtual_method_table` array in the same way as non-abstract methods.

#### `interfaces[]`

The `interfaces` item of the `class_info` structure represents a table of `implemented_interface_info` structures. The table must contain an entry for each

of the interfaces indicated in the declaration of this class and each of the interfaces in the hierarchies of those interfaces. Interfaces that occur more than once are represented by a single entry. Interfaces implemented by superclasses of this class may optionally be represented.

Given the declarations below, the number of entries for class c0 is 1 and the entry in the interfaces array is i0. The minimum number of entries for class c1 is 3 and the entries in the interfaces array are i1, i2, and i3. The entries class c1 may also include interface i0, which is implemented by the superclass of c1.

```
interface i0 {}
interface i1 {}
interface i2 extends i1 {}
interface i3 {}
class c0 implements i0 {}
class c1 extends c0 implements i2, i3 {}
```

The `implemented_interface_info` structure is defined as follows:

```
implemented_interface_info {
    class_ref interface
    u1 count
    u1 index[count]
}
```

#### interface

The interface item has the form of a `class_ref` structure. The `class_ref` structure is defined as part of the `CONSTANT_Classref_info` structure (“`CONSTANT_Classref`” on page 64). The `interface_info` structure referenced by the interface item represents an interface implemented by this class.

#### count

The count item indicates the number of entries in the index array.

#### index

The index item is an array that maps declarations of interface methods to the implementations in this class. It is a representation of a all of the methods declared by the interface and its superinterfaces. Entries in the index array must be ordered such that the token of the declared interface method is equal to the index. The token value is that assigned to the method within the scope of the interface definition and its superinterfaces, not within the scope of this class.

The values in the `index` array represent the tokens of the implementation of the interface method. This token value is in the scope of this class.

---

## 6.8 Method Component

The Method Component describes each of the methods declared in this package, except `<clinit>` methods and interface method declarations. The exception handlers associated with each method are also described. The Method Component is represented by the following structure:

```
method_component {
    u1 tag
    u2 size
    u1 handlers_count
    exception_handler_info exception_handlers[handlers_count]
    method_info methods[]
}
```

The items in the `method_component` structure are as follows:

**tag**

The `tag` item has the value `COMPONENT_Method(7)`.

**size**

The `size` item indicates the number of bytes in the `method_component` structure, excluding the `tag` and `size` items.

**handlers\_count**

The `handlers_count` item represents the number of entries in the `exception_handlers` array.

**exception\_handlers[]**

The `exception_handlers` item represents an array of 8-byte `exception_handler_info` structures. Each `exception_handler_info` structure represents a *catch or finally* block defined in a method of this package.

Entries in the `exception_handlers` array are sorted in ascending order by the distance between the beginning of the Method Component to the endpoint of each active exception handler range in the `methods` item.

**methods[]**

The `methods` item represents an array of variable-length `method_info` structures.

Each entry represents a method declared in a class or interface of this package. *<clinit>* methods and interface method declaration are not included; all other abstract methods are.

## 6.8.1 exception\_handler\_info

The `exception_handler_info` structure is defined as follows:

```
exception_handler_info {  
    u2 start_offset  
    u2 active_length  
    u2 handler_offset  
    u2 catch_type_index  
}
```

The items in the `exception_handler_info` structure are as follows:

### start\_offset, active\_length

The `active_length` item is encoded to indicate whether the active range of this exception handler is nested within another exception handler. The high bit of the `active_length` is equal to 1 if the active range is not contained within another exception handler, and this exception handler is the last handler applicable to the active range. The high bit is equal to 0 if the active range is contained within the active range of another exception handler, or there are successive handlers applicable to the same active range.

`end_offset` is defined as `start_offset` plus `active_length` & 0x7FFF.

The `start_offset` item and `end_offset` are byte offsets into the info item of the Method Component. They indicate the ranges in a bytecode array at which the exception handler is active. The value of the `start_offset` must be a valid offset into a bytecode array to the opcode of an instruction. The value of the `end_offset` either must be a valid offset into a code array of the opcode of an instruction or must be equal to a method's bytecode count, the length of the code array. The value of the `start_offset` must be less than the value of the `end_offset`.

The `start_offset` is inclusive and the `end_offset` is exclusive; that is, the exception handler must be active while the execution address is within the interval (`start_offset`, `end_offset`).

### handler\_offset

The `handler_offset` item represents a byte offset into the info item of the Method Component. It indicates the start of the exception handler. The value of the item

must be a valid offset into a method's bytecode array to an opcode of an instruction, and must be less than the value of the method's bytecode count.

#### catch\_type\_index

If the value of the catch\_type\_index item is non-zero, it must be a valid index into the constant\_pool table. The constant\_pool entry at that index must be a CONSTANT\_Classref\_info structure, representing the class of the exception caught by this exception\_handlers table entry.

If the exception\_handlers table entry represents a finally block, the value of the catch\_type\_index item is zero. In this case the exception handler is called for all exceptions that are thrown within the start\_offset and end\_offset range.

## 6.8.2 method\_info

The method\_info structure is defined as follows:

```
method_info {
    method_header_info method_header
    u1 bytecodes[]
}
```

The items in the method\_info structure are as follows:

#### method\_header

The method\_header item represents either a method\_header\_info or extended\_method\_header\_info structure:

```
method_header_info {
    u1 bitfield {
        bit[4] flags
        bit[4] max_stack
    }
    u1 bitfield {
        bit[4] nargs
        bit[4] max_locals
    }
}

extended_method_header_info {
    u1 bitfield {
        bit[4] flags
        bit[4] padding
    }
    u1 max_stack
}
```

```

        u1 nargs
        u1 max_locals
    }

```

The items of the `method_header_info` and `extended_method_header_info` structures are as follows:

### flags

The `flags` item is a mask of modifiers defined for the method. Valid flag values are shown in the following table.

TABLE 6-6 Method flags

Flags	Values
ACC_EXTENDED	0x8
ACC_ABSTRACT	0x4

The value of the `ACC_EXTENDED` flag must be one if the `method_header` is represented by an `extended_method_header_info` structure. Otherwise the value must be zero.

The value of the `ACC_ABSTRACT` flag must be one if this method is defined as abstract. In this case the `bytecodes` array must be empty. If the method is not abstract the value of the `ACC_ABSTRACT` flag must be zero.

The Java Card Virtual Machine reserves all other flag values. Their values must be zero.

### padding

The `padding` item has the value of zero. This item is only defined for the `extended_method_header_info` structure.

### max\_stack

The `max_stack` item indicates the maximum number of 16-bit cells required on the operand stack during execution of this method.

Stack entries of type *int* are represented in two 16-bit cells, while all others are represented in one 16-bit cell.

### nargs

The `nargs` item indicates the number of 16-bit cells required to represent the parameters passed to this method, including the *this* pointer for non-static methods.

Parameters of type *int* are represented in two 16-bit cells, while all others are rep-

resented in one 16-bit cell.

#### max\_locals

The `max_locals` item indicates the number of 16-bit cells required to the local variables declared by this method, not including the parameters passed to the method on invocation.

---

**Note** – Unlike in Java Card CAP files, in Java `class` files `max_locals` includes both the local variables declared in a method and the parameters passed to a method.

---

Local variables of type `int` are represented in two 16-bit cells, while all others are represented in one 16-bit cell. The number of cells required for overloaded local variables is two if one or more of the overloaded variables is of type `int`.

#### code[]

The `code` item represents an array of Java Card bytecodes that implement the method.

---

## 6.9 Static Field Component

The Static Field Component contains all of the information required to create and initialize an image of the static fields defined in this package. Final static fields of primitive types are not represented in this component. Instead these compile-time constants occur inline in Java Card instructions.

The ordering constraints associated with the static field image are as shown in the following figure. Reference-types occur first in the image. Arrays initialized through Java `<clinit>` methods occur first, and primitive types initialized to non-default values occur last.

reference types	arrays of primitive types initialized by <code>&lt;clinit&gt;</code> methods
	reference types initialized to null
primitive types	primitive types initialized to default values
	primitive types initialized to non-default values

FIGURE 6-1 Static Field Order Map

The number of bytes used to represent each field type in the static field image is shown in the following table.

TABLE 6-7 Static Field Sizes

Type	Bytes
boolean	1
byte	1
short	2
int	4
reference	2

The `static_field_component` structure is defined as:

```
static_field_component {
    u1 tag
    u2 size
    u2 byte_count
    u2 reference_count
    u2 array_init_count
    array_init_info array_init[array_init_count]
    u2 default_value_count
    u2 non_default_value_count
    u1 non_default_values[non_default_values_count]
}
```

The items in the `static_field_component` structure are as follows:

**tag**

The `tag` item has the value `COMPONENT_StaticField(8)`.

**size**

The `size` item indicates the number of bytes in the `static_field_component` structure, excluding the `tag` and `size` items.

**byte\_count**

The `byte_count` item indicates the number of bytes required to represent the image of the static fields defined in this package. The number of bytes required to represent each type is shown in the previous table.

The value of the `byte_count` item does not include the number of bytes required to represent the array instances defined in the Static Field Component.

**reference\_count**

The `reference_count` item indicates the number of reference-type static fields defined in this package.

### array\_init\_count

The `array_init_count` item indicates the number of elements in the `array_init` array. In CAP files that define a library package the value of `array_init_count` must be zero.

### array\_init[]

The `array_init` item represents an array of `array_init_info` structures that specify the initial array values of final static fields of arrays of primitive types. These initial values are indicated in Java `<clinit>` methods. The `array_init_info` structure is defined as:

```
array_init_info {  
    u1 type  
    u2 count  
    u1 values[count]  
}
```

The `type` item represents the type of the primitive array. Valid values are shown in the following table.

TABLE 6-8 Array Types

Type	Value
boolean	0
byte	1
short	2
int	4

The `count` item represents the number of elements in the `values` array.

This value of the `count` item is not the same as the number of elements in the array of the final static field. The number of elements in the array is equal to the `count` item divided by the number of bytes required to represent the static field type (TABLE 6-7) indicated by the `type` item.

The `values` array represents a byte array representing the initial values of the static field array.

### default\_value\_count

The `default_value_count` item indicates the number of bytes required to initialize the set of static fields that are to be initialized to default values. The number of bytes required for each static field type is equal to the size in bytes of the type is shown in TABLE 6-7.

**non\_default\_value\_count**

The **non\_default\_value\_count** item represents the number elements in the **non\_default\_values** array.

of bytes of initial values primitive-type static fields initialized to non-default values. This value is number of elements in the **non\_default\_values** array item.

**non\_default\_values[]**

The **non\_default\_values** item represents an array of bytes of non-default initial values. The number of entries in the array for each static field type is equal to the size in bytes of the type as shown in TABLE 6-7.

---

## 6.10 Reference Location Component

The Reference Location Component represents lists of offsets into the Method Component to operands that contain indices into the **constant\_pool** table. Some of the indices are represented in one-byte values while other are represented in two-byte values. The structure is defined as:

```
reference_location_component {  
    u1 tag  
    u2 size  
    u2 byte_index_count  
    u1 offsets_to_byte_indices[byte_index_count]  
    u2 byte2_index_count  
    u1 offsets_to_byte2_indices[byte2_index_count]  
}
```

The items of the **reference\_location\_component** structure are as follows:

**tag**

The tag item has the value **COMPONENT\_ReferenceLocation(9)**.

**size**

The size item indicates the number of bytes in the **reference\_location\_component** structure, excluding the tag and size items.

**byte\_index\_count**

The **byte\_index\_count** item represents the number of elements in the **offsets\_to\_byte\_indices** array.

#### offsets\_to\_byte\_indices[]

The `offsets_to_byte_indices` item represents an array of 1-byte jump offsets into the `info` item of the Method Component to each 1-byte `constant_pool` table index. Each entry represents the number of bytes (or *distance*) between the previous index to the next. If the distance is greater than or equal to 255 then there are *n* entries equal of 255 in the array, where *n* is equal to the distance divided by 255. The *n*th entry of 255 is followed by an entry containing the value of distance modulo 255. An example of the jump offsets in an `offsets_to_byte_indices` array, given a set of offsets to 1-byte constant pool references, is shown in Table 6-9, "One-byte Reference Location Example".

TABLE 6-9 One-byte Reference Location Example

Instruction	Offset to Operand	Jump Offset
getfield_a 0	10	10
putfield_b 2	65	55
		255
		255
getfield_s 1	580	5
		255
putfield_a 0	835	0
getfield_i 3	843	8

All one-byte `constant_pool` table indices in the Method Component must be represented in `offsets_to_byte_indices` array.

#### byte2\_index\_count

The `byte2_index_count` item represents the number of elements in the `offsets_to_byte2_indices` array.

#### offsets\_to\_byte2\_indices[]

The `offsets_to_byte2_indices` item represents an array of 1-byte jump offsets into the `info` item of the Method Component to each 2-byte `constant_pool` table index. Each entry represents the number of bytes (or *distance*) between the previous index to the next. If the distance is greater than or equal to 255 then there are *n* entries equal of 255 in the array where *n* is equal to the distance divided by 255. The *n*th entry of 255 is followed by an entry containing the value of distance modulo 255.

An example of the jump offsets in an `offsets_to_byte2_indices` array, given a set of offsets to 1-byte constant pool references, is shown in Table 6-9, "One-byte Reference Location Example". The same example applies to the

offsets\_to\_byte2\_indices table if the instructions are changed to those with 2-byte constant pool indices.

All two-byte constant\_pool table indices in the Method Component must be represented in offsets\_to\_byte2\_indices array, including those represented in catch\_type\_index items of the exception\_handler\_info table.

---

## 6.11 Export Component

The Export Component describes all elements in this package that other packages may reference directly. If this CAP file does not contain an Applet Component, the Export Component includes entries for all public classes. Furthermore, for each public class it includes entries for all public and protected static methods, and all public and protected static fields. The set of static fields does not include primitive final static fields (compile-time constants). Packages that reference instance fields and non-special virtual methods in this package do so indirectly through class references and field or method tokens, and are therefore not represented in the Export Component.

If CAP file contains an Applet Component, the Export Component includes only entries for all public interfaces that are shareable. An interface is shareable if and only if it is the *javacard.framework.Sharable* interface or implements (directly or indirectly) that interface.

---

**Note** – The restriction on exportable functionality is imposed by the firewall as defined in the *Java Card Runtime Environment 2.1* specifications.

---

The Export Component is represented by the following structure:

```
export_component {
    u1 tag
    u2 size
    u2 class_count
    class_export_info {
        u2 class_offset
        u1 static_field_count
        u1 static_method_count
        u2 static_field_offsets[static_field_count]
        u2 static_method_offsets[static_method_count]
    } class_exports[class_count]
}
```

The items of the export\_component structure are as follows:

tag

The tag item has the value `COMPONENT_Export(10)`.

size

The size item indicates the number of bytes in the `export_component` structure, excluding the tag and size items.

class\_count

The `class_count` item represents the number of public classes in this package. It indicates the number of elements in the `class_exports` array.

class\_exports[]

The `class_exports` item represents a variable-length table of `class_export_info` structures. The table contains an entry for each of the public classes defined in this package. An index into the table to a particular element must be equal to the token value of the class.

The items in the `class_export_info` structure are:

class\_offset

The `class_offset` item represents a byte offset into the info item of the Class Component to a `class_info` structure. The `class_info` structure at that offset must represent the exported class.

static\_field\_count

The `static_field_count` item represents the number of elements in the `static_field_offsets` array. This value indicates the number of public and protected static fields defined in this class, excluding final static fields of primitive types.

static\_method\_count

The `static_method_count` item represents the number of elements in the `static_method_offsets` array. This value indicates the number of public and protected static methods and constructors defined in this class.

static\_field\_offsets[]

The `static_field_offset` item represents an array of 2-byte offsets to a static field in the static field image defined by the Static Field Component. The static field defined at each offset must represent the exported field with the token value equal to the index to the entry in the `static_field_offsets` array.

`static_method_offsets[]`

The `static_method_offset` item represents a table of 2-byte offsets into the `info` item of the Method Component to a `method_info` structure. The `method_info` structure at each offset must represent the exported method with the token value equal to the index to the entry in the `static_method_offsets` array.

---

## 6.12 Descriptor Component

The Descriptor Component provides sufficient information to parse and verify all elements of the CAP file. The Descriptor Component is represented by the following structure:

```
descriptor_component {
    u1 tag
    u2 size
    u2 class_count
    class_descriptor_info classes[class_count]
    type_descriptor_info types
}
```

The items of the `descriptor_component` structure are as follows:

**tag**

The `tag` item has the value `COMPONENT_Descriptor(11)`.

**size**

The `size` item indicates the number of bytes in the `descriptor_component` structure, excluding the `tag` and `size` items.

**class\_count**

The `class_count` item represents the number of `class_descriptor_info` structures in the `classes` array.

**classes[]**

The `classes` item represents a table of variable-length `class_descriptor_info` structures. Each class and interface defined in this package is represented in the table.

**types**

The `types` item represents a `type_descriptor_info` structure.

## 6.12.1 class\_descriptor\_info

The `class_descriptor_info` structure is used to describe a Java class or interface:

```
class_descriptor_info {  
    u1 token  
    u1 access_flags  
    class_ref this_class_ref  
    u1 interface_count  
    u2 field_count  
    u2 method_count  
    class_ref interfaces [interface_count]  
    field_descriptor_info fields[field_count]  
    method_descriptor_info methods[method_count]  
}
```

The items of the `class_descriptor_info` structure are as follows:

### token

The `token` item represents the class token of this class or interface. If this class is private or package-visible it does not have a token assigned. In this case the value of the `token` item must be 0xFF.

### access\_flags

The `access_flags` item is a mask of modifiers used to describe the access permission to and properties of this class. The `access_flags` modifiers for classes are shown in the following table.

TABLE 6-10 Class Access and Modifier Flags

Name	Value
ACC_PUBLIC	0x01
ACC_FINAL	0x10
ACC_INTERFACE	0x40
ACC_ABSTRACT	0x80

The Java Card Virtual Machine reserves all other flag values. Their values must be zero.

### this\_class\_ref

The `this_class_ref` item is a `class_ref` structure indicating the location of the `class_info` structure in the Class Component ("Class Component" on page 70). The `class_ref` structure is defined as part of the `CONSTANT_Classref_info` structure ("CONSTANT\_Classref" on page 64).

### interface\_count

The `interface_count` item represents the number of interfaces implemented by

this class.

#### field\_count

The `field_count` item represents the number of `field_descriptor_info` structures in the `fields` array. It is the number of fields defined in this class.

#### method\_count

The `method_count` item represents the number of `method_descriptor_info` structures in the `methods` array. It is the number of methods declared or defined in this class or interface.

#### interfaces []

The `interface_refs` item represents an array of interfaces implemented by this class. The elements in the array are `class_ref` structures indicating the location of the `class_info` structure in the Class Component ("Class Component" on page 70). The `class_ref` structure is defined as part of the `CONSTANT_Classref_info` structure ("CONSTANT\_Classref" on page 64).

#### fields[]

The `fields` item represents a table of variable-length `field_descriptor_info` structures.

#### methods[]

The `methods` item represents a table of variable-length `method_descriptor_info` structures.

## 6.12.2 field\_descriptor\_info

The `field_descriptor_info` structure is used to describe a Java field:

```
field_descriptor_info {
    u1 token
    u1 access_flags
    union {
        static_field_ref static_field
        instance_field_ref instance_field
    } field_ref
    union {
        u2 primitive_type
        u2 type_offset
    } type
}
```

The items of the `field_descriptor_info` structure is as follows:

## token

The token item represents the token of this field. If this field is private or package-visible static field it does not have a token assigned. In this case the value of the token item must be 0xFF.

## access\_flags

The access\_flags item is a mask of modifiers used to describe the access permission to and properties of the field. The access\_flags modifiers for instance fields are shown in the following table.

TABLE 6-11 Field Access and Modifier Flags

Name	Value
ACC_PUBLIC	0x01
ACC_PRIVATE	0x02
ACC_PROTECTED	0x04
ACC_STATIC	0x08
ACC_FINAL	0x10

All of the access flags are the same as those defined in a Java class file.

The Java Card Virtual Machine reserves all other flag values. Their values must be zero.

## field\_ref

The field\_ref item represents a reference to the field. If the ACC\_STATIC flag is equal to 1, this item represents a static\_field\_ref as defined in the CONSTANT\_StaticFieldref structure ("CONSTANT\_StaticFieldref and CONSTANT\_StaticMethodref" on page 68). If the ACC\_STATIC flag is equal to 0, this item represents an instance\_field\_ref as defined in the CONSTANT\_InstanceFieldref structure ("CONSTANT\_InstanceFieldref, CONSTANT\_VirtualMethodref, and CONSTANT\_SuperMethodref" on page 66).

## type

The type item represents the type of the field, directly or indirectly. If the field is a primitive type (*boolean*, *byte*, *short*, or *int*) the high bit of this item is set and the type is specified using the values in the following table. Otherwise the type item represents a 15-bit offset into the type\_descriptor table. The item at the offset in the type\_descriptor table must represent the type of the field.

TABLE 6-12 Primitive Type Descriptor Values

Data Type	Value
boolean	0x02
byte	0x03
short	0x04
integer	0x05

### 6.12.3 `method_descriptor_info`

The `method_descriptor_info` structure is used to describe a Java method:

```
method_descriptor_info {
    u1 token
    u1 access_flags
    u2 method_offset
    u2 type_offset
    u2 bytecode_count
    u2 exception_handler_count
    u2 exception_handler_index
}
```

The items of the `method_descriptor_info` structure are as follows:

#### `token`

The `token` item represents the token of this method. If this method is a private or package-visible static or constructor method, or private virtual method it does not have a token assigned. In this case the value of the `token` item must be 0xFF.

#### `access_flags`

The `access_flags` item is a mask of modifiers used to describe the access permission to and properties of the method. The `access_flags` modifiers for methods are shown in the following table.

**TABLE 6-13 Method Access and Modifier Flags**

Name	Value
ACC_PUBLIC	0x01
ACC_PRIVATE	0x02
ACC_PROTECTED	0x04
ACC_STATIC	0x08
ACC_FINAL	0x10
ACC_ABSTRACT	0x40
ACC_INIT	0x80

All of the access flags are the same as those defined in a Java class file except the ACC\_INIT flag.

The ACC\_INIT flag is set if the method descriptor identifies a constructor methods. In Java a constructor method is recognize by the class file verifier by its name, <init>, but in Java Card the name is replaced by a token. These methods require special checks by a verifier.

The Java Card Virtual Machine reserves all other flag values. Their values must be zero.

#### method\_offset

The method\_offset item represents a byte offset into the info item of the Method Component ("Method Component" on page 76) to a method\_info structure. The method\_info structure must represent this method.

If the class\_descriptor\_info structure that contains this method\_descriptor\_info structure represents an interface, the value of the method\_offset item must be zero.

#### type\_offset

The type\_offset item must be a valid offset into the type\_descriptor\_info array. The type described at that offset represents the signature of the method.

#### bytecode\_count

The bytecode\_count item represents the number of bytecodes in the method.

#### exception\_handler\_count

The exception\_handler\_count item represents the number of exception handler implemented by this method.

#### exception\_handler\_index

The exception\_handler\_index item represents the index to the first

exception\_handlers table entry of the method\_info structure that is implemented by this method. Succeeding exception\_handlers table entries, up to the value of the exception\_handler\_count item, are also exception handlers implemented by this method.

## 6.12.4 type\_descriptor\_info

The type\_descriptor\_info structure represents the types of fields and signatures of methods as follows:

```
type_descriptor_info {
    u2 cp_count
    u2 constant_types[cp_types]
    { u1 nibble_count;
      u1 type[(nibble_count+1) / 2];
    } type_desc[]
}
```

The structure type\_descriptor\_info structure contains the following elements:

### cp\_count

The cp\_count value represents the number of entries in the constant\_pool table.

### constant\_types[]

The constant\_types item represents an array of offsets to type\_descriptor structures corresponding to the parallel entries in the constant\_pool table. If the corresponding constant\_pool table entry does not have an associated type, the value of the entry in the constant\_types array item is 0xFFFF.

### type\_desc[]

The type\_desc item represents an array of variable-length type descriptor structures. These descriptors represent the types of fields and signatures of methods. The elements in the structure are:

### nibble\_count

The nibble\_count value represents the number of nibbles in the type array.

### type[]

The type array contains an encoded description of the type composed of individual nibbles. If nibble\_count item is an odd number, the last nibble must be 0x0. The values of the type descriptor nibbles are defined in the following table.

TABLE 6-14 Type Descriptor Values

Type	Value
void	0x1
boolean	0x2
byte	0x3
short	0x4
int	0x5
objectref	0x6
array of boolean	0xA
array of byte	0xB
array of short	0xC
array of int	0xD
array of objectref	0xE

Class reference types are described using the class nibble 0x6, followed by a 2-byte (4-nibble) `class_ref` structure. The `class_ref` structure is defined as part of the `CONSTANT_Classref_info` structure ("CONSTANT\_Classref" on page 64). For example, a field of type reference to `p1.cl` in a CAP file defining package `p0` is described as:

0x6	<p1>	<c1>	0x0
objectref	package token (high bit on)	class token	padding

The following are examples of the array-types:

byte[]	0xB	0x0		
	array of byte	padding		
p1.cl[]	0xE	<p1>	<c1>	0x0
	array of objectref	package token (high bit on)	class token	padding

Method signatures are encoded in the same way, with the last nibble indicating the return type of the method, for example:

()V	0x1	0x0		
	void	padding		
(lp1.cl;)S	0x6	<p1>	<c1>	0x4
	objectref	package token (high bit on)	class token	short

## Java Card Virtual Machine Instruction Set

---

A Java Card Virtual Machine instruction consists of an opcode specifying the operation to be performed, followed by zero or more operands embodying values to be operated upon. This chapter gives details about the format of each Java Card Virtual Machine instruction and the operation it performs.

---

### 7.1 Assumptions: The Meaning of “Must”

The description of each instruction is always given in the context of Java Card Virtual Machine code that satisfies the static and structural constraints of Chapter 6, “The Cap File Format.”

In the description of individual Java Card Virtual Machine instructions, we frequently state that some situation “must” or “must not” be the case: “The *value2* must be of type *int*.” The constraints of Chapter 6, “The Cap File Format” guarantee that all such expectations will in fact be met. If some constraint (a “must” or “must not”) in an instruction description is not satisfied at run time, the behavior of the Java Card Virtual Machine is undefined.

---

### 7.2 Reserved Opcodes

In addition to the opcodes of the instructions specified later this chapter, which are used in Java Card CAP files (see Chapter 5, “The Cap File Format”), two opcodes are reserved for internal use by a Java Card Virtual Machine implementation. If Sun extends the instruction set of the Java Card Virtual Machine in the future, these reserved opcodes are guaranteed not to be used.

The two reserved opcodes, numbers 254 (0xfe) and 255 (0xff), have the mnemonics *impdep1* and *impdep2*, respectively. These instructions are intended to provide “back doors” or traps to implementation-specific functionality implemented in software and hardware, respectively.

Although these opcodes have been reserved, they may only be used inside a Java Card Virtual Machine implementation. They cannot appear in valid CAP files.

---

## 7.3 Virtual Machine Errors

A Java Card Virtual Machine throws an object that is an instance of a subclass of the class `VirtualMachineError` when an internal error or resource limitation prevents it from implementing the semantics of the Java Language. The Java Card Virtual Machine specification cannot predict where resource limitations or internal errors may be encountered and does not mandate precisely when they can be reported. Thus, any of the virtual machine errors listed as subclasses of `VirtualMachineError` in Section 2.3.3.4, “Errors,” on page 2-20 may be thrown at any time during the operation of the Java Card Virtual Machine.

---

## 7.4 Security Exceptions

Instructions of the Java Card Virtual Machine throw an instance of the class `SecurityException` when a security violation has been detected. The Java Card Virtual Machine does not mandate the complete set of security violations which can or will result in an exception being thrown. However, there is a minimum set which must be supported.

In the general case, any instruction which de-references an object reference must throw a `SecurityException` if the applet context in which the instruction is executing is different than the owning applet context of the referenced object. The list of instructions includes the instance field get and put instructions, the array load and store instructions, as well as the *arraylength*, *invokeinterface*, *invokespecial*, *invokevirtual*, *checkcast*, *instanceof* and *athrow* instructions.

There are several exceptions to this general rule that allow cross-context use of objects or arrays. These exceptions are detailed in Chapter 6 of the *Java Card Runtime Environment Specification*. An important detail to note is that any cross-context method invocation will result in a context switch.

The Java Card Virtual Machine may also throw a `SecurityException` if an instruction violates any of the static constraints of Chapter 6, “The Cap File Format.” The Java Card Virtual Machine specification does not mandate which instructions must implement these additional security checks, or to what level. Therefore, a `SecurityException` may be

thrown at any time during the operation of the Java Card Virtual Machine.

## 7.5 The Java Card Virtual Machine Instruction Set

Java Virtual Machine instructions are represented in this chapter by entries of the form shown in the figure below, an example instruction page, in alphabetical order and each beginning on a new page.

<b><i>mnemonic</i></b>	Short description of the instruction
<b>Format</b>	<i>mnemonic</i> <i>operand1</i> <i>operand2</i> ...
<b>Forms</b>	<i>mnemonic</i> = opcode
<b>Stack</b>	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>value3</i>
<b>Description</b>	A longer description detailing constraints on operand stack contents or constant pool entries, the operation performed, the type of the results, etc.
<b>Runtime Exceptions</b>	If any runtime exceptions can be thrown by the execution of an instruction they are set off one to a line, in the order in which they must be thrown.  Other than the runtime exceptions, if any, listed for an instruction, that instruction must not throw any runtime exceptions except for instances of <code>VirtualMachineError</code> or its subclasses.
<b>Notes</b>	Comments not strictly part of the specification of an instruction are set aside as notes at the end of the description.

FIGURE 7-1 An example instruction page

Each cell in the instruction format diagram represents a single 8-bit byte. The instruction's *mnemonic* is its name. Its opcode is its numeric representation and is given in both decimal

and hexadecimal forms. Only the numeric representation is actually present in the Java Card Virtual Machine code in a CAP file.

Keep in mind that there are “operands” generated at compile time and embedded within Java Card Virtual Machine instructions, as well as “operands” calculated at run time and supplied on the operand stack. Although they are supplied from several different areas, all these operands represent the same thing: values to be operated upon by the Java Card Virtual Machine instruction being executed. By implicitly taking many of its operands from its operand stack, rather than representing them explicitly in its compiled code as additional operand bytes, register numbers, etc., the Java Card Virtual Machine’s code stays compact.

Some instructions are presented as members of a family of related instructions sharing a single description, format, and operand stack diagram. As such, a family of instructions includes several opcodes and opcode mnemonics; only the family mnemonic appears in the instruction format diagram, and a separate forms line lists all member mnemonics and opcodes. For example, the forms line for the *sconst\_<s>* family of instructions, giving mnemonic and opcode information for the two instructions in that family (*sconst\_0* and *sconst\_1*), is

**Forms**   *sconst\_0* = 3 (0x3),  
              *sconst\_1* = 4 (0x4)

In the description of the Java Card Virtual Machine instructions, the effect of an instruction’s execution on the operand stack of the current frame is represented textually, with the stack growing from left to right and each word represented separately. Thus,

**Stack...**, *value1*, *value2* ⇒  
..., *result*

shows an operation that begins by having a one-word *value2* on top of the operand stack with a one-word *value1* just beneath it. As a result of the execution of the instruction, *value1* and *value2* are popped from the operand stack and replaced by a one-word *result*, which has been calculated by the instruction. The remainder of the operand stack, represented by an ellipsis (...), is unaffected by the instruction’s execution.

The type *int* takes two words on the operand stack. In the operand stack representation, each word is represented separately using a dot notation:

**Stack...**, *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2* ⇒  
..., *result.word1*, *result.word2*

The Java Card Virtual Machine specification does not mandate how the two words are used to represent the 32-bit *int* value; it only requires that a particular implementation be internally consistent.

## **aaload**

Load reference from array

### **Format**

<i>aaload</i>
---------------

### **Forms**

*aaload* = 36 (0x24)

### **Stack**

..., *arrayref*, *index*  $\Rightarrow$   
..., *value*

### **Description**

The *arrayref* must be of type reference and must refer to an array whose components are of type reference. The *index* must be of type short. Both *arrayref* and *index* are popped from the operand stack. The reference *value* in the component of the array at *index* is retrieved and pushed onto the top of the operand stack.

### **Runtime Exceptions**

If *arrayref* is null, *aaload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *aaload* instruction throws an `ArrayIndexOutOfBoundsException`.

## aastore

Store into reference array

### Format

<i>aastore</i>
----------------

### Forms

*aastore* = 55 (0x37)

### Stack

..., *arrayref*, *index*, *value* ⇒  
...

### Description

The *arrayref* must be of type *reference* and must refer to an array whose components are of type *reference*. The *index* must be of type *short* and the *value* must be of type *reference*. The *arrayref*, *index* and *value* are popped from the operand stack. The *reference value* is stored as the component of the array at *index*.

The type of value must be assignment compatible with the type of the components of the array referenced by *arrayref*. Assignment of a value of reference type *S* (source) to a variable of reference type *T* (target) is allowed only when the type *S* supports all of the operations defined on type *T*. The detailed rules follow:

- If *S* is a class type, then:
  - If *T* is a class type, then *S* must be the same class as *T*, or *S* must be a subclass of *T*,
  - If *T* is an interface type, *S* must implement interface *T*.
- If *S* is an array type, namely the type *SC*[], that is, an array of components of type *SC*, then:
  - If *T* is a class type, *T* must be *Object*, or:
    - If *T* is an array type, namely the type *TC*[], an array of components of type *TC*, then either *TC* and *SC* must be the same primitive type, or
    - *TC* and *SC* must both be reference types with type *SC* assignable to *TC*, by these rules.

*S* cannot be an interface type, because there are no instances of interfaces, only instances of classes and arrays.

### Runtime Exceptions

If *arrayref* is null, *aastore* throws a *NullPointerException*.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *aastore* instruction throws an *ArrayIndexOutOfBoundsException*.

Otherwise, if *arrayref* is not null and the actual type of *value* is not assignment compatible with the actual type of the component of the array, *aastore* throws an *ArrayStoreException*.

## **aconst\_null**

Push null

### **Format**

<i>aconst_null</i>
--------------------

### **Forms**

*aconst\_null* = 1 (0x1)

### **Stack**

...  $\Rightarrow$   
..., *null*

### **Description**

Push the null object reference onto the operand stack.

## aload

Load reference from local variable

### Format

<i>aload</i>
<i>index</i>

### Forms

*aload* = 21 (0x15)

### Stack

...  $\Rightarrow$   
..., *objectref*

### Description

The *index* is an unsigned byte that must be a valid index into the local variables of the current frame. The local variable at *index* must contain a reference. The *objectref* in the local variable at *index* is pushed onto the operand stack.

### Notes

The *aload* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction is intentional.

## **aload\_<n>**

Load reference from local variable

### **Format**

<i>aload_&lt;n&gt;</i>
------------------------

### **Forms**

*aload\_0* = 24 (0x18)

*aload\_1* = 25 (0x19)

*aload\_2* = 26 (0x1a)

*aload\_3* = 27 (0x1b)

### **Stack**

...  $\Rightarrow$

..., *objectref*

### **Description**

The <n> must be a valid index into the local variables of the current frame. The local variable at <n> must contain a reference. The *objectref* in the local variable at <n> is pushed onto the operand stack.

### **Notes**

An *aload\_<n>* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the corresponding *astore\_<n>* instruction is intentional.

Each of the *aload\_<n>* instructions is the same as *aload* with an *index* of <n>, except that the operand <n> is implicit.

## **anewarray**

Create new array of reference

### **Format**

<i>anewarray</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

### **Forms**

*anewarray* = 145 (0x91)

### **Stack**

..., *count* ⇒

..., *arrayref*

### **Description**

The *count* must be of type short. It is popped off the operand stack. The *count* represents the number of components of the array to be created. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package, where the value of the index is  $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$ . The item at that index in the constant pool must be of type `CONSTANT_Classref`, a reference to a class or interface type. The reference is resolved. A new array with components of that type, of length *count*, is allocated from the heap, and a reference *arrayref* to this new array object is pushed onto the operand stack. All components of the new array are initialized to null, the default value for reference types.

### **Runtime Exception**

If *count* is less than zero, the *anewarray* instruction throws a `NegativeArraySizeException`.

## **areturn**

Return reference from method

### **Format**

<i>areturn</i>
----------------

### **Forms**

*areturn* = 119 (0x77)

### **Stack**

..., *objectref* ⇒  
[empty]

### **Description**

The *objectref* must be of type `reference`. The *objectref* is popped from the operand stack of the current frame and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The virtual machine then reinstates the frame of the invoker and returns control to the invoker.

## arraylength

Get length of array

### Format

<i>arraylength</i>
--------------------

### Forms

*arraylength* = 146 (0x92)

### Stack

..., *arrayref* ⇒

..., *length*

### Description

The *arrayref* must be of type `reference` and must refer to an array. It is popped from the operand stack. The *length* of the array it references is determined. That *length* is pushed onto the top of the operand stack as a short.

### Runtime Exception

If *arrayref* is null, the *arraylength* instruction throws a `NullPointerException`.

## **astore**

Store reference into local variable

### **Format**

<i>astore</i>
<i>index</i>

### **Forms**

*astore* = 40 (0x28)

### **Stack**

..., *objectref* ⇒  
...

### **Description**

The *index* is an unsigned byte that must be a valid index into the local variables of the current frame. The *objectref* on the top of the operand stack must be of type `returnAddress` or of type `reference`. The *objectref* is popped from the operand stack, and the value of the local variable at *index* is set to *objectref*.

### **Notes**

The *astore* instruction is used with an *objectref* of type `returnAddress` when implementing Java's `finally` keyword. The *aload* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction is intentional.

## **astore\_<n>**

Store reference into local variable

### **Format**

<i>astore_&lt;n&gt;</i>
-------------------------

### **Forms**

*astore\_0* = 43 (0x2b)

*astore\_1* = 44 (0x2c)

*astore\_2* = 45 (0x2d)

*astore\_3* = 46 (0x2e)

### **Stack**

..., *objectref* ⇒

...

### **Description**

The <*n*> must be a valid index into the local variables of the current frame. The *objectref* on the top of the operand stack must be of type `returnAddress` or of type `reference`. It is popped from the operand stack, and the value of the local variable at <*n*> is set to *objectref*.

### **Notes**

An *astore\_<n>* instruction is used with an *objectref* of type `returnAddress` when implementing Java's `finally` keyword. An *aload\_<n>* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the corresponding *astore\_<n>* instruction is intentional.

Each of the *aload\_<n>* instructions is the same as *aload* with an *index* of <*n*>, except that the operand <*n*> is implicit.

## athrow

Throw exception or error

### Format

<i>athrow</i>
---------------

### Forms

*athrow* = 147 (0x93)

### Stack

..., *objectref* ⇒  
*objectref*

### Description

The *objectref* must be of type reference and must refer to an object which is an instance of class Throwable or of a subclass of Throwable. It is popped from the operand stack. The *objectref* is then thrown by searching the current frame for the most recent catch clause that catches the class of *objectref* or one of its superclasses.

If a catch clause is found, it contains the location of the code intended to handle this exception. The pc register is reset to that location, the operand stack of the current frame is cleared, *objectref* is pushed back onto the operand stack, and execution continues. If no appropriate clause is found in the current frame, that frame is popped, the frame of its invoker is reinstated, and the *objectref* is rethrown.

If no catch clause is found that handles this exception, the virtual machine exits.

### Runtime Exception

If *objectref* is null, *athrow* throws a NullPointerException instead of *objectref*.

## **baload**

Load byte or boolean from array

### **Format**

<i>baload</i>
---------------

### **Forms**

*baload* = 37 (0x25)

### **Stack**

..., *arrayref*, *index*  $\Rightarrow$   
..., *value*

### **Description**

The *arrayref* must be of type reference and must refer to an array whose components are of type byte or of type boolean. The *index* must be of type short. Both *arrayref* and *index* are popped from the operand stack. The byte *value* in the component of the array at *index* is retrieved, sign-extended to a short *value*, and pushed onto the top of the operand stack.

### **Runtime Exceptions**

If *arrayref* is null, *aaload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *aaload* instruction throws an `ArrayIndexOutOfBoundsException`.

## **bastore**

Store into byte or boolean array

### **Format**

<i>bastore</i>
----------------

### **Forms**

*bastore* = 56 (0x38)

### **Stack**

..., *arrayref*, *index*, *value* ⇒  
...

### **Description**

The *arrayref* must be of type reference and must refer to an array whose components are of type byte or of type boolean. The *index* and *value* must both be of type short. The *arrayref*, *index* and *value* are popped from the operand stack. The short *value* is truncated to a byte and stored as the component of the array indexed by *index*.

### **Runtime Exceptions**

If *arrayref* is null, *bastore* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *bastore* instruction throws an `ArrayIndexOutOfBoundsException`.

## bipush

Push byte

### Format

<i>bipush</i>
<i>byte</i>

### Forms

*bipush* = 18 (0x12)

### Stack

...  $\Rightarrow$

..., *value.word1*, *value.word2*

### Description

The immediate *byte* is sign-extended to an `int`, and the resulting *value* is pushed onto the operand stack.

### Notes

If a virtual machine does not support the `int` data type, the *bipush* instruction will not be available.

## bspush

Push byte

### Format

<i>bspush</i>
<i>byte</i>

### Forms

*bspush* = 16 (0x10)

### Stack

...  $\Rightarrow$   
..., *value*

### Description

The immediate *byte* is sign-extended to a short, and the resulting *value* is pushed onto the operand stack.

## checkcast

Check whether object is of given type

### Format

<i>checkcast</i>
<i>atype</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

### Forms

*checkcast* = 148 (0x94)

### Stack

..., *objectref* ⇒  
..., *objectref*

### Description

The unsigned byte *atype* is a code that indicates if the type against which the object is being checked is an array type or a class type. It must take one of the following values or zero:

Array Type	<i>atype</i>
T_BOOLEAN	10
T_BYTE	11
T_SHORT	12
T_INT	13
T_REFERENCE	14

If the value of *atype* is 10, 11, 12, or 13, the values of the *indexbyte1* and *indexbyte2* must be zero, and the value of *atype* indicates the array type against which to check the object. Otherwise the unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package, where the value of the index is  $(\text{indexbyte1} \ll 8) | \text{indexbyte2}$ . The item at that index in the constant pool must be of type `CONSTANT_Classref`, a reference to a class or interface type. The reference is resolved. If the value of *atype* is 14, the object is checked against an array type which is an array of object references of the type of the resolved class. If the value of *atype* is zero, the object is checked against a class or interface type which is the resolved class.

The *objectref* must be of type reference. If *objectref* is null or can be cast to the specified array type or the resolved class or interface type, the operand stack is unchanged; otherwise the *checkcast* instruction throws a `ClassCastException`.

The following rules are used to determine whether an *objectref* that is not null can be cast to the resolved type: if *S* is the class of the object referred to by *objectref* and *T* is the resolved class, array or interface type, *checkcast* determines whether *objectref* can be cast to type *T* as follows:

- If *S* is an ordinary (non-array) class, then:
  - If *T* is a class type, then *S* must be the same class as *T*, or a subclass of *T*.
  - If *T* is an interface type, *S* must implement interface *T*.

- S** cannot be an interface type, because there are no instances of interfaces, only instances of classes and arrays.

If *objectref* cannot be cast to the resolved class, array, or interface type, the *checkcast* instruction throws a *ClassCastException*.

The *checkcast* instruction is fundamentally very similar to the *instanceof* instruction. It differs in its treatment of null, its behavior when its test fails (*checkcast* throws an exception, *instanceof* pushes a result code), and its effect on the operand stack.

Chapter 7 Java Card Virtual Machine Instruction Set 115

## dup

Duplicate top operand stack word

### Format

<i>dup</i>
------------

### Forms

*dup* = 61 (0x3d)

### Stack

..., *word*  $\Rightarrow$   
..., *word*, *word*

### Description

The top word on the operand stack is duplicated and pushed onto the operand stack.

The *dup* instruction must not be used unless *word* contains a 16-bit data type.

### Notes

Except for restrictions preserving the integrity of 32-bit data types, the *dup* instruction operates on an untyped word, ignoring the type of data it contains.

## dup\_x

Duplicate top operand stack words and insert below

### Format

<i>dup_x</i>
<i>mn</i>

### Forms

*dup\_x* = 63 (0x3f)

### Stack

..., wordN, ..., wordM, ..., word1  $\Rightarrow$   
..., wordM, ..., word1, wordN, ..., wordM, ..., word1

### Description

The unsigned byte *mn* is used to construct two parameter values. The high nybble, (*mn* & 0xf0) >> 4, is used as the value *m*. The low nybble, (*mn* & 0xf), is used as the value *n*. Permissible values for *m* are 1 through 4. Permissible values for *n* are 0 and *m* through *m*+4.

For positive values of *n*, the top *m* words on the operand stack are duplicated and the copied words are inserted *n* words down in the operand stack. When *n* equals 0, the top *m* words are copied and placed on top of the stack.

The *dup\_x* instruction must not be used unless the ranges of words 1 through *m* and words *m*+1 through *n* each contain either a 16-bit data type, two 16-bit data types, a 32-bit data type, a 16-bit data type and a 32-bit data type (in either order), or two 32-bit data types.

### Notes

Except for restrictions preserving the integrity of 32-bit data types, the *dup\_x* instruction operates on untyped words, ignoring the types of data they contain.

If a virtual machine does not support the `int` data type, the permissible values for *m* are 1 or 2, and permissible values for *n* are 0 and *m* through *m*+2.

## dup2

Duplicate top two operand stack words

### Format

<i>dup2</i>
-------------

### Forms

*dup2* = 62 (0x3e)

### Stack

..., *word2*, *word1*  $\Rightarrow$   
..., *word2*, *word1*, *word2*, *word1*

### Description

The top two words on the operand stack are duplicated and pushed onto the operand stack, in the original order.

The *dup2* instruction must not be used unless each of *word1* and *word2* is a word that contains a 16-bit data type or both together are the two words of a single 32-bit datum.

### Notes

Except for restrictions preserving the integrity of 32-bit data types, the *dup2* instruction operates on untyped words, ignoring the types of data they contain.

## getfield\_<t>

Fetch field from object

### Format

getfield_<t>
index

### Forms

getfield\_a = 131 (0x83)  
getfield\_b = 132 (0x84)  
getfield\_s = 133 (0x85)  
getfield\_i = 134 (0x86)

### Stack

..., objectref ⇒  
..., value  
  
OR  
  
..., objectref ⇒  
..., value.word1, value.word2

### Description

The *objectref*, which must be of type reference, is popped from the operand stack. The unsigned *index* is used as an index into the constant pool of the current package. The constant pool item at the index must be of type CONSTANT\_InstanceFieldref, a reference to a class and a field token. If the field is protected, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type reference
- *b* field must be of type byte or type boolean
- *s* field must be of type short
- *i* field must be of type int

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The *value* at that offset into the class instance referenced by *objectref* is fetched. If the *value* is of type byte or type boolean, it is sign-extended to a short. The *value* is pushed onto the operand stack.

### Runtime Exception

If *objectref* is null, the *getfield\_<t>* instruction throws a NullPointerException.

### Notes

If a virtual machine does not support the int data type, the *getfield\_i* instruction will not be available.

## getfield\_<t>\_this

Fetch field from current object

### Format

getfield_<t>_this
index

### Forms

getfield\_a\_this = 173 (0xad)  
getfield\_b\_this = 174 (0xae)  
getfield\_s\_this = 175 (0xaf)  
getfield\_i\_this = 176 (0xb0)

### Stack

... ⇒  
..., value  
  
OR  
  
... ⇒  
..., value.word1, value.word2

### Description

The currently executing method must be an instance method. The local variable at index 0 must contain a reference *objectref* to the currently executing method's *this* parameter. The unsigned *index* is used as an index into the constant pool of the current package. The constant pool item at the index must be of type `CONSTANT_InstanceFieldref`, a reference to a class and a field token. If the field is protected, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type reference
- *b* field must be of type byte or type boolean
- *s* field must be of type short
- *i* field must be of type int

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The *value* at that offset into the class instance referenced by *objectref* is fetched. If the *value* is of type byte or type boolean, it is sign-extended to a short. The *value* is pushed onto the operand stack.

### Runtime Exception

If *objectref* is null, the `getfield_<t>_this` instruction throws a `NullPointerException`.

### Notes

If a virtual machine does not support the `int` data type, the `getfield_i_this` instruction will not be available.

## getfield\_<t>\_w

Fetch field from object (wide index)

### Format

<i>getfield_&lt;t&gt;_w</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

### Forms

*getfield\_a\_w* = 169 (0xa9)  
*getfield\_b\_w* = 170 (0xaa)  
*getfield\_s\_w* = 171 (0xab)  
*getfield\_i\_w* = 172 (0xac)

### Stack

..., *objectref* ⇒  
..., *value*  
  
OR  
  
..., *objectref* ⇒  
..., *value.word1*, *value.word2*

### Description

The *objectref*, which must be of type reference, is popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package, where the value of the index is  $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$ . The constant pool item at the index must be of type `CONSTANT_InstanceFieldref`, a reference to a class and a field token. The item must resolve to a field of type reference. If the field is protected, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type reference
- *b* field must be of type byte or type boolean
- *s* field must be of type short
- *i* field must be of type int

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The *value* at that offset into the class instance referenced by *objectref* is fetched. If the *value* is of type byte or type boolean, it is sign-extended to a short. The *value* is pushed onto the operand stack.

### Runtime Exception

If *objectref* is null, the *getfield\_<t>\_w* instruction throws a `NullPointerException`.

### Notes

If a virtual machine does not support the `int` data type, the *getfield\_i\_w* instruction will not

be available.

## getstatic\_<t>

Get static field from class

### Format

<i>getstatic_&lt;t&gt;</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

### Forms

*getstatic\_a* = 123 (0x7b)  
*getstatic\_b* = 124 (0x7c)  
*getstatic\_s* = 125 (0x7d)  
*getstatic\_i* = 126 (0x7e)

### Stack

...  $\Rightarrow$   
..., *value*

OR

...  $\Rightarrow$   
..., *value.word1*, *value.word2*

### Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package, where the value of the index is  $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$ . The constant pool item at the index must be of type `CONSTANT_StaticFieldref`, a reference to a static field. If the field is protected, then it must be either a member of the current class or a member of a superclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type reference
- *b* field must be of type byte or type boolean
- *s* field must be of type short
- *i* field must be of type int

The width of a class field is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The item is resolved, determining the class field. The *value* of the class field is fetched. If the *value* is of type byte or type boolean, it is sign-extended to a short. The *value* is pushed onto the operand stack.

### Notes

If a virtual machine does not support the `int` data type, the *getstatic\_i* instruction will not be available.

## goto

Branch always

### Format

<i>goto</i>
<i>branch</i>

### Forms

*goto* = 112 (0x70)

### Stack

No change

### Description

The value *branch* is used as a signed 8-bit offset. Execution proceeds at that offset from the address of the opcode of this *goto* instruction. The target address must be that of an opcode of an instruction within the method that contains this *goto* instruction.

## **goto\_w**

Branch always (wide index)

### **Format**

<i>goto_w</i>
<i>branchbyte1</i>
<i>branchbyte2</i>

### **Forms**

*goto\_w* = 168 (0xa8)

### **Stack**

No change

### **Description**

The unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is  $(branchbyte1 \ll 8) \mid branchbyte2$ . Execution proceeds at that offset from the address of the opcode of this *goto* instruction. The target address must be that of an opcode of an instruction within the method that contains this *goto* instruction.

## **i2b**

Convert int to byte

### **Format**

<i>i2b</i>
------------

### **Forms**

*i2b* = 93 (0x5d)

### **Stack**

..., *value.word1*, *value.word2* ⇒  
..., *result*

### **Description**

The *value* on top of the operand stack must be of type `int`. It is popped from the operand stack and converted to a `byte` *result* by taking the low-order 16 bits of the `int` value, and discarding the high-order 16 bits. The low-order word is truncated to a `byte`, then sign-extended to a `short` *result*. The *result* is pushed onto the operand stack.

### **Notes**

The *i2b* instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign as *value*.

If a virtual machine does not support the `int` data type, the *i2b* instruction will not be available.

## **i2s**

Convert int to short

### **Format**

<i>i2s</i>
------------

### **Forms**

*i2s* = 94 (0x5e)

### **Stack**

..., *value.word1*, *value.word2* ⇒  
..., *result*

### **Description**

The *value* on top of the operand stack must be of type `int`. It is popped from the operand stack and converted to a short *result* by taking the low-order 16 bits of the `int` value and discarding the high-order 16 bits. The *result* is pushed onto the operand stack.

### **Notes**

The *i2s* instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign as *value*.

If a virtual machine does not support the `int` data type, the *i2s* instruction will not be available.

## **iadd**

Add int

### **Format**

<i>iadd</i>
-------------

### **Forms**

*iadd* = 66 (0x42)

### **Stack**

..., *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2* ⇒  
..., *result.word1*, *result.word2*

### **Description**

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

If an *iadd* instruction overflows, then the result is the low-order bits of the true mathematical result in a sufficiently wide two's-complement format. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

### **Notes**

If a virtual machine does not support the `int` data type, the *iadd* instruction will not be available.

## **iaload**

Load int from array

### **Format**

<i>iaload</i>
---------------

### **Forms**

*iaload* = 39 (0x27)

### **Stack**

..., *arrayref*, *index*  $\Rightarrow$   
..., *value.word1*, *value.word2*

### **Description**

The *arrayref* must be of type reference and must refer to an array whose components are of type int. The *index* must be of type short. Both *arrayref* and *index* are popped from the operand stack. The int *value* in the component of the array at *index* is retrieved and pushed onto the top of the operand stack.

### **Runtime Exceptions**

If *arrayref* is null, *aaload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *aaload* instruction throws an `ArrayIndexOutOfBoundsException`.

### **Notes**

If a virtual machine does not support the int data type, the *iaload* instruction will not be available.

## **iand**

Boolean AND int

### **Format**

<i>iand</i>
-------------

### **Forms**

*iand* = 84 (0x54)

### **Stack**

..., *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2* ⇒  
..., *result.word1*, *result.word2*

### **Description**

Both *value1* and *value2* must be of type `int`. They are popped from the operand stack. An `int` *result* is calculated by taking the bitwise AND (conjunction) of *value1* and *value2*. The *result* is pushed onto the operand stack.

### **Notes**

If a virtual machine does not support the `int` data type, the *iand* instruction will not be available.

## **iastore**

Store into int array

### **Format**

<i>iastore</i>
----------------

### **Forms**

*iastore* = 58 (0x3a)

### **Stack**

..., *arrayref*, *index*, *value.word1*, *value.word2* ⇒

...

### **Description**

The *arrayref* must be of type reference and must refer to an array whose components are of type int. The *index* must be of type short and *value* must be of type int. The *arrayref*, *index* and *value* are popped from the operand stack. The int *value* is stored as the component of the array indexed by *index*.

### **Runtime Exception**

If *arrayref* is null, *sastore* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *sastore* instruction throws an `ArrayIndexOutOfBoundsException`.

### **Notes**

If a virtual machine does not support the int data type, the *iastore* instruction will not be available.

## **icmp**

Compare int

### **Format**

<i>icmp</i>
-------------

### **Forms**

*icmp* = 95 (0x5f)

### **Stack**

..., *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2* ⇒  
..., *result*

### **Description**

Both *value1* and *value2* must be of type `int`. They are both popped from the operand stack, and a signed integer comparison is performed. If *value1* is greater than *value2*, the short value 1 is pushed onto the operand stack. If *value1* is equal to *value2*, the short value 0 is pushed onto the operand stack. If *value1* is less than *value2*, the short value -1 is pushed onto the operand stack.

### **Notes**

If a virtual machine does not support the `int` data type, the *icmp* instruction will not be available.

## **iconst\_<i>**

Push int constant

### **Format**

<i>iconst_&lt;i&gt;</i>
-------------------------

### **Forms**

*iconst\_m1* = 10 (0x09)

*iconst\_0* = 11 (0xa)

*iconst\_1* = 12 (0xb)

*iconst\_2* = 13 (0xc)

*iconst\_3* = 14 (0xd)

*iconst\_4* = 15 (0xe)

*iconst\_5* = 16 (0xf)

### **Stack**

... ⇒

..., <i>.word1, <i>.word2

### **Description**

Push the int constant <i> (-1, 0, 1, 2, 3, 4, or 5) onto the operand stack.

### **Notes**

If a virtual machine does not support the int data type, the *iconst\_<i>* instruction will not be available.

## **idiv**

Divide int

### **Format**

<i>idiv</i>
-------------

### **Forms**

*idiv* = 72 (0x48)

### **Stack**

..., *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2*  $\Rightarrow$   
..., *result.word1*, *result.word2*

### **Description**

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is the value of the Java expression *value1* / *value2*. The *result* is pushed onto the operand stack.

An `int` division rounds towards 0; that is, the quotient produced for `int` values in *n*/*d* is an `int` value *q* whose magnitude is as large as possible while satisfying  $|d \cdot q| = |n|$ . Moreover, *q* is a positive when  $|n| = |d|$  and *n* and *d* have the same sign, but *q* is negative when  $|n| = |d|$  and *n* and *d* have opposite signs.

There is one special case that does not satisfy this rule: if the dividend is the negative integer of the largest possible magnitude for the `int` type, and the divisor is `-1`, then overflow occurs, and the result is equal to the dividend. Despite the overflow, no exception is thrown in this case.

### **Runtime Exception**

If the value of the divisor in an `int` division is 0, *idiv* throws an `ArithmeticException`.

### **Notes**

If a virtual machine does not support the `int` data type, the *idiv* instruction will not be available.

## **if\_acmp<cond>**

Branch if reference comparison succeeds

### **Format**

<i>if_acmp&lt;cond&gt;</i>
<i>branch</i>

### **Forms**

*if\_acmpeq* = 104 (0x68)

*if\_acmpne* = 105 (0x69)

### **Stack**

..., *value1*, *value2* ⇒

...

### **Description**

Both *value1* and *value2* must be of type `reference`. They are both popped from the operand stack and compared. The results of the comparisons are as follows:

- *eq* succeeds if and only if *value1* = *value2*
- *ne* succeeds if and only if *value1* ≠ *value2*

If the comparison succeeds, *branch* is used as signed 8-bit offset, and execution proceeds at that offset from the address of the opcode of this *if\_acmp<cond>* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if\_acmp<cond>* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if\_acmp<cond>* instruction.

## **if\_acmp<cond>\_w**

Branch if reference comparison succeeds (wide index)

### **Format**

<i>if_acmp&lt;cond&gt;_w</i>
<i>branchbyte1</i>
<i>branchbyte2</i>

### **Forms**

*if\_acmpeq\_w* = 160 (0xa0)

*if\_acmpne\_w* = 161 (0xa1)

### **Stack**

..., *value1*, *value2* ⇒

...

### **Description**

Both *value1* and *value2* must be of type `reference`. They are both popped from the operand stack and compared. The results of the comparisons are as follows:

- *eq* succeeds if and only if *value1* = *value2*
- *ne* succeeds if and only if *value1* ≠ *value2*

If the comparison succeeds, the unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is  $(branchbyte1 \ll 8) \mid branchbyte2$ . Execution proceeds at that offset from the address of the opcode of this *if\_acmp<cond>\_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if\_acmp<cond>\_w* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if\_acmp<cond>\_w* instruction.

## **if\_scmp<cond>**

Branch if short comparison succeeds

### **Format**

<i>if_scmp&lt;cond&gt;</i>
<i>branch</i>

### **Forms**

*if\_scmpeq* = 106 (0x6a)  
*if\_scmpne* = 107 (0x6b)  
*if\_scmplt* = 108 (0x6c)  
*if\_scmpge* = 109 (0x6d)  
*if\_scmpgt* = 110 (0x6e)  
*if\_scmple* = 111 (0x6f)

### **Stack**

..., *value1*, *value2* ⇒  
...

### **Description**

Both *value1* and *value2* must be of type `short`. They are both popped from the operand stack and compared. All comparisons are signed. The results of the comparisons are as follows:

- *eq* succeeds if and only if *value1* = *value2*
- *ne* succeeds if and only if *value1* ≠ *value2*
- *lt* succeeds if and only if *value1* < *value2*
- *le* succeeds if and only if *value1* ≤ *value2*
- *gt* succeeds if and only if *value1* > *value2*
- *ge* succeeds if and only if *value1* ≥ *value2*

If the comparison succeeds, *branch* is used as signed 8-bit offset, and execution proceeds at that offset from the address of the opcode of this *if\_scmp<cond>* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if\_scmp<cond>* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if\_scmp<cond>* instruction.

## **if\_scmp<cond>\_w**

Branch if short comparison succeeds (wide index)

### **Format**

<i>if_scmp&lt;cond&gt;_w</i>
<i>branchbyte1</i>
<i>branchbyte2</i>

### **Forms**

*if\_scmpeq\_w* = 162 (0xa2)  
*if\_scmpne\_w* = 163 (0xa3)  
*if\_scmplt\_w* = 164 (0xa4)  
*if\_scmpge\_w* = 165 (0xa5)  
*if\_scmpgt\_w* = 166 (0xa6)  
*if\_scmple\_w* = 167 (0xa7)

### **Stack**

..., *value1*, *value2* ⇒  
...

### **Description**

Both *value1* and *value2* must be of type short. They are both popped from the operand stack and compared. All comparisons are signed. The results of the comparisons are as follows:

- *eq* succeeds if and only if *value1* = *value2*
- *ne* succeeds if and only if *value1* ≠ *value2*
- *lt* succeeds if and only if *value1* < *value2*
- *le* succeeds if and only if *value1* ≤ *value2*
- *gt* succeeds if and only if *value1* > *value2*
- *ge* succeeds if and only if *value1* ≥ *value2*

If the comparison succeeds, the unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is (*branchbyte1* << 8) | *branchbyte2*. Execution proceeds at that offset from the address of the opcode of this *if\_scmp<cond>\_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if\_scmp<cond>\_w* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if\_scmp<cond>\_w* instruction.

## if<cond>

Branch if short comparison with zero succeeds

### Format

<i>if&lt;cond&gt;</i>
<i>branch</i>

### Forms

*ifeq* = 96 (0x60)  
*ifne* = 97 (0x61)  
*iflt* = 98 (0x62)  
*ifge* = 99 (0x63)  
*ifgt* = 100 (0x64)  
*ifle* = 101 (0x65)

### Stack

..., *value* ⇒  
...

### Description

The *value* must be of type short. It is popped from the operand stack and compared against zero. All comparisons are signed. The results of the comparisons are as follows:

- *eq* succeeds if and only if *value* = 0
- *ne* succeeds if and only if *value* ≠ 0
- *lt* succeeds if and only if *value* < 0
- *le* succeeds if and only if *value* ≤ 0
- *gt* succeeds if and only if *value* > 0
- *ge* succeeds if and only if *value* ≥ 0

If the comparison succeeds, *branch* is used as signed 8-bit offset, and execution proceeds at that offset from the address of the opcode of this *if<cond>* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if<cond>* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if<cond>* instruction.

## **if<cond>\_w**

Branch if short comparison with zero succeeds (wide index)

### **Format**

<i>if&lt;cond&gt;_w</i>
<i>branchbyte1</i>
<i>branchbyte2</i>

### **Forms**

*ifeq\_w* = 152 (0x98)  
*ifne\_w* = 153 (0x99)  
*iflt\_w* = 154 (0x9a)  
*ifge\_w* = 155 (0x9b)  
*ifgt\_w* = 156 (0x9c)  
*ifle\_w* = 157 (0x9d)

### **Stack**

..., *value* ⇒

...

### **Description**

The *value* must be of type short. It is popped from the operand stack and compared against zero. All comparisons are signed. The results of the comparisons are as follows:

- *eq* succeeds if and only if *value* = 0
- *ne* succeeds if and only if *value* ≠ 0
- *lt* succeeds if and only if *value* < 0
- *le* succeeds if and only if *value* ≤ 0
- *gt* succeeds if and only if *value* > 0
- *ge* succeeds if and only if *value* ≥ 0

If the comparison succeeds, the unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is  $(branchbyte1 \ll 8) \mid branchbyte2$ . Execution proceeds at that offset from the address of the opcode of this *if<cond>\_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if<cond>\_w* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if<cond>\_w* instruction.

## **ifnonnull**

Branch if reference not null

### **Format**

<i>ifnonnull</i>
<i>branch</i>

### **Forms**

*ifnonnull* = 103 (0x67)

### **Stack**

..., *value* ⇒  
...

### **Description**

The *value* must be of type *reference*. It is popped from the operand stack. If the *value* is not null, *branch* is used as signed 8-bit offset, and execution proceeds at that offset from the address of the opcode of this *ifnonnull* instruction. The target address must be that of an opcode of an instruction within the method that contains this *ifnonnull* instruction.

Otherwise, execution proceeds at the address of the instruction following this *ifnonnull* instruction.

## **ifnonnull\_w**

Branch if reference not null (wide index)

### **Format**

<i>ifnonnull_w</i>
<i>branchbyte1</i>
<i>branchbyte2</i>

### **Forms**

*ifnonnull\_w* = 159 (0x9f)

### **Stack**

..., *value* ⇒

...

### **Description**

The *value* must be of type `reference`. It is popped from the operand stack. If the *value* is not null, the unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is  $(branchbyte1 \ll 8) \mid branchbyte2$ . Execution proceeds at that offset from the address of the opcode of this *ifnonnull\_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *ifnonnull\_w* instruction.

Otherwise, execution proceeds at the address of the instruction following this *ifnonnull\_w* instruction.

## **`ifnull`**

Branch if reference is null

### **Format**

<i>ifnull</i>
<i>branch</i>

### **Forms**

*ifnull* = 102 (0x66)

### **Stack**

..., *value* ⇒  
...

### **Description**

The *value* must be of type reference. It is popped from the operand stack. If the *value* is null, *branch* is used as signed 8-bit offset, and execution proceeds at that offset from the address of the opcode of this *ifnull* instruction. The target address must be that of an opcode of an instruction within the method that contains this *ifnull* instruction.

Otherwise, execution proceeds at the address of the instruction following this *ifnull* instruction.

## **ifnull\_w**

Branch if reference is null (wide index)

### **Format**

<i>ifnull_w</i>
<i>branchbyte1</i>
<i>branchbyte2</i>

### **Forms**

*ifnull\_w* = 158 (0x9e)

### **Stack**

..., *value* ⇒

...

### **Description**

The *value* must be of type `reference`. It is popped from the operand stack. If the *value* is `null`, the unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is  $(branchbyte1 \ll 8) \mid branchbyte2$ . Execution proceeds at that offset from the address of the opcode of this *ifnull\_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *ifnull\_w* instruction.

Otherwise, execution proceeds at the address of the instruction following this *ifnull\_w* instruction.

## **iinc**

Increment local `int` variable by constant

### **Format**

<i>iinc</i>
<i>index</i>
<i>const</i>

### **Forms**

*iinc* = 90 (0x5a)

### **Stack**

No change

### **Description**

The *index* is an unsigned byte. Both *index* and *index* + 1 must be valid indices into the local variables of the current frame. The local variables at *index* and *index* + 1 together must contain an `int`. The *const* is an immediate signed byte. The value *const* is first sign-extended to an `int`, then the `int` contained in the local variables at *index* and *index* + 1 is incremented by that amount.

### **Notes**

If a virtual machine does not support the `int` data type, the *iinc* instruction will not be available.

## **iinc\_w**

Increment local int variable by constant

### **Format**

<i>iinc_w</i>
<i>index</i>
<i>byte1</i>
<i>byte2</i>

### **Forms**

*iinc\_w* = 151 (0x97)

### **Stack**

No change

### **Description**

The *index* is an unsigned byte. Both *index* and *index* + 1 must be valid indices into the local variables of the current frame. The local variables at *index* and *index* + 1 together must contain an int. The immediate unsigned *byte1* and *byte2* values are assembled into an intermediate short where the value of the short is  $(\text{byte1} \ll 8) \mid \text{byte2}$ . The intermediate value is then sign-extended to an int *const*. The int contained in the local variables at *index* and *index* + 1 is incremented by *const*.

### **Notes**

If a virtual machine does not support the int data type, the *iinc\_w* instruction will not be available.

## **iipush**

Push int

### **Format**

<i>iipush</i>
<i>byte1</i>
<i>byte2</i>
<i>byte3</i>
<i>byte4</i>

### **Forms**

*iipush* = 20 (0x14)

### **Stack**

...  $\Rightarrow$   
..., *value1.word1*, *value1.word2*

### **Description**

The immediate unsigned *byte1*, *byte2*, *byte3*, and *byte4* values are assembled into a signed int where the value of the int is  $(byte1 \ll 24) \mid (byte2 \ll 16) \mid (byte3 \ll 8) \mid byte4$ . The resulting *value* is pushed onto the operand stack.

### **Notes**

If a virtual machine does not support the int data type, the *iipush* instruction will not be available.

## **iload**

Load int from local variable

### **Format**

<i>iload</i>
<i>index</i>

### **Forms**

*iload* = 23 (0x17)

### **Stack**

...  $\Rightarrow$   
..., *value1.word1*, *value1.word2*

### **Description**

The *index* is an unsigned byte. Both *index* and *index* + 1 must be valid indices into the local variables of the current frame. The local variables at *index* and *index* + 1 together must contain an int. The *value* of the local variables at *index* and *index* + 1 is pushed onto the operand stack.

### **Notes**

If a virtual machine does not support the int data type, the *iload* instruction will not be available.

## **iload\_<n>**

Load int from local variable

### **Format**

<i>iload_&lt;n&gt;</i>
------------------------

### **Forms**

*iload\_0* = 32 (0x20)

*iload\_1* = 33 (0x21)

*iload\_2* = 34 (0x22)

*iload\_3* = 35 (0x23)

### **Stack**

... ⇒

..., *value1.word1*, *value1.word2*

### **Description**

Both <*n*> and <*n*> + 1 must be a valid indices into the local variables of the current frame. The local variables at <*n*> and <*n*> + 1 together must contain an int. The *value* of the local variables at <*n*> and <*n*> + 1 is pushed onto the operand stack.

### **Notes**

Each of the *iload\_<n>* instructions is the same as *iload* with an *index* of <*n*>, except that the operand <*n*> is implicit.

If a virtual machine does not support the int data type, the *iload\_<n>* instruction will not be available.

## ilookupswitch

Access jump table by key match and jump

### Format

<i>ilookupswitch</i>
<i>defaultbyte1</i>
<i>defaultbyte2</i>
<i>npairs1</i>
<i>npairs2</i>
<i>match-offset pairs...</i>

### Pair Format

<i>matchbyte1</i>
<i>matchbyte2</i>
<i>matchbyte3</i>
<i>matchbyte4</i>
<i>offsetbyte1</i>
<i>offsetbyte2</i>

### Forms

*ilookupswitch* = 118 (0x76)

### Stack

..., *key.word1*, *key.word2* ⇒  
...

### Description

An *ilookupswitch* instruction is a variable-length instruction. Immediately after the *ilookupswitch* opcode follow a signed 16-bit value *default*, an unsigned 16-bit value *npairs*, and then *npairs* pairs. Each pair consists of an *int match* and a signed 16-bit *offset*. Each *match* is constructed from four unsigned bytes as  $(\text{matchbyte1} \ll 24) | (\text{matchbyte2} \ll 16) | (\text{matchbyte3} \ll 8) | \text{matchbyte4}$ . Each *offset* is constructed from two unsigned bytes as  $(\text{offsetbyte1} \ll 8) | \text{offsetbyte2}$ .

The table *match-offset* pairs of the *ilookupswitch* instruction must be sorted in increasing numerical order by *match*.

The *key* must be of type *int* and is popped from the operand stack and compared against the *match* values. If it is equal to one of them, then a target address is calculated by adding the corresponding *offset* to the address of the opcode of this *ilookupswitch* instruction. If the *key* does not match any of the *match* values, the target address is calculated by adding *default* to the address of the opcode of this *ilookupswitch* instruction. Execution then continues at the target address.

The target address that can be calculated from the offset of each *match-offset* pair, as well as the one calculated from *default*, must be the address of an opcode of an instruction within the method that contains this *ilookupswitch* instruction.

### Notes

The *match-offset* pairs are sorted to support lookup routines that are quicker than linear

search.

If a virtual machine does not support the `int` data type, the *ilookupswitch* instruction will not be available.

## **imul**

Multiply int

### **Format**

<i>imul</i>
-------------

### **Forms**

*imul* = 70 (0x46)

### **Stack**

..., *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2*  $\Rightarrow$   
..., *result.word1*, *result.word2*

### **Description**

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* \* *value2*. The *result* is pushed onto the operand stack.

If an *imul* instruction overflows, then the result is the low-order bits of the mathematical product as an `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical product of the two values.

### **Notes**

If a virtual machine does not support the `int` data type, the *imul* instruction will not be available.

## ineg

Negate int

### Format

<i>ineg</i>
-------------

### Forms

*ineg* = 76 (0x4c)

### Stack

..., *value.word1*, *value.word2*  $\Rightarrow$   
..., *result.word1*, *result.word2*

### Description

The *value* must be of type `int`. It is popped from the operand stack. The `int` *result* is the arithmetic negation of *value*,  $-value$ . The *result* is pushed onto the operand stack.

For `int` values, negation is the same as subtraction from zero. Because the Java Card Virtual Machine uses two's-complement representation for integers and the range of two's-complement values is not symmetric, the negation of the maximum negative `int` results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown.

For all `int` values  $x$ ,  $-x$  equals  $(\sim x) + 1$ .

### Notes

If a virtual machine does not support the `int` data type, the *imul* instruction will not be available.

## instanceof

Determine if object is of given type

### Format

<i>instanceof</i>
<i>atype</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

### Forms

*instanceof* = 149 (0x95)

### Stack

..., *objectref* ⇒  
..., *result*

### Description

The unsigned byte *atype* is a code that indicates if the type against which the object is being checked is an array type or a class type. It must take one of the following values or zero:

Array Type	<i>atype</i>
T_BOOLEAN	10
T_BYTE	11
T_SHORT	12
T_INT	13
T_REFERENCE	14

If the value of *atype* is 10, 11, 12, or 13, the values of the *indexbyte1* and *indexbyte2* must be zero, and the value of *atype* indicates the array type against which to check the object. Otherwise the unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package, where the value of the index is  $(\text{indexbyte1} \ll 8) | \text{indexbyte2}$ . The item at that index in the constant pool must be of type

CONSTANT\_Classref, a reference to a class or interface type. The reference is resolved. If the value of *atype* is 14, the object is checked against an array type which is an array of object references of the type of the resolved class. If the value of *atype* is zero, the object is checked against a class or interface type which is the resolved class.

The *objectref* must be of type reference. It is popped from the operand stack. If *objectref* is not null and is an instance of the resolved class, array or interface, the *instanceof* instruction pushes a short *result* of 1 on the operand stack. Otherwise it pushes a short *result* of 0.

The following rules are used to determine whether an *objectref* that is not null is an instance of the resolved type: if *S* is the class of the object referred to by *objectref* and *T* is the resolved class, array or interface type, *instanceof* determines whether *objectref* is an instance of *T* as follows:

- If *S* is an ordinary (non-array) class, then:
  - If *T* is a class type, then *S* must be the same class as *T*, or a subclass of *T*.

- If *T* is an interface type, *S* must implement interface *T*.
- If *S* is a class representing the array type *SC*[], that is, an array of components of type *SC*, then:
  - If *T* is a class type, then *T* must be `Object`.
  - If *T* is an array type *TC*[], that is, an array of components of type *TC*, then one of the following must be true:
    - *TC* and type *SC* are the same primitive type.
    - *TC* and *SC* are reference types, and type *SC* can be cast to *TC* by these runtime rules.

*S* cannot be an interface type, because there are no instances of interfaces, only instances of classes and arrays.

#### Notes

The *instanceof* instruction is fundamentally very similar to the *checkcast* instruction. It differs in its treatment of `null`, its behavior when its test fails (*checkcast* throws an exception, *instanceof* pushes a result code), and its effect on the operand stack.

If a virtual machine does not support the `int` data type, the value of *atype* may not be 13 (array type = `T_INT`).

## invokeinterface

Invoke interface method

### Format

<i>invokeinterface</i>
<i>nargs</i>
<i>indexbyte1</i>
<i>indexbyte2</i>
<i>method</i>

### Forms

*invokeinterface* = 142 (0x8e)

### Stack

..., *objectref*, [*arg1*, [*arg2* ...]] ⇒  
...

### Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package, where the value of the index is  $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$ . The constant pool item at that index must be of type `CONSTANT_Classref`, a reference to an interface class. The specified method is resolved. The interface method must not be `<init>`, an instance initialization method, or `<clinit>`, a class or interface initialization method.

The *nargs* operand is an unsigned byte which must not be zero. The *method* operand is an unsigned byte which is the interface method token for the method to be invoked. The *objectref* must be of type reference and must be followed on the operand stack by *nargs* – 1 words of arguments. The number of words of arguments and the type and order of the values they represent must be consistent with those of the selected interface method.

The interface table of the class of the type of *objectref* is determined. If *objectref* is an array type, then the interface table of class `Object` is used. The interface table is searched for the resolved interface. The result of the search is a table which is used to map the *method* token to a *index*.

The *index* is an unsigned byte which is used as an index into the method table of the class of the type of *objectref*. If the *objectref* is an array type, then the method table of class `Object` is used. The table entry at that index includes a direct reference to the method's code and modifier information.

The *nargs* – 1 words of arguments and *objectref* are popped from the operand stack. A new stack frame is created for the method being invoked, and *objectref* and the arguments are made the values of its first *nargs* words of local variables, with *arg1* in local variable at index 0, *arg1* in local variable at offset 2, *arg2* immediately following that, and so on. The new stack frame is then made current, and the Java Card Virtual Machine *pc* is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

**Runtime Exception**

If *objectref* is null, the *invokeinterface* instruction throws a `NullPointerException`.

## invokespecial

Invoke instance method; special handling for superclass, private, and instance initialization method invocations

### Format

<i>invokespecial</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

### Forms

*invokespecial* = 140 (0x8c)

### Stack

..., *objectref*, [*arg1*, [*arg2* ...]] ⇒  
...

### Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package, where the value of the index is  $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$ . The constant pool item at that index must be of type `CONSTANT_StaticMethodref`, a reference to a statically linked instance method, or of type `CONSTANT_SuperMethodref`, a reference to an instance method of a specified class. The reference is resolved. The resolved method must not be `<clinit>`, a class or interface initialization method. If the method is `<init>`, an instance initialization method, then the method must only be invoked once on an uninitialized object, and before the first backward branch following the execution of the *new* instruction that allocated the object. Finally, if the method is protected, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The resolved method includes the code for the method, an unsigned byte *nargs* that must not be zero, and the method's modifier information.

The *objectref* must be of type `reference`, and must be followed on the operand stack by *nargs* – 1 words of arguments, where the number of words of arguments and the type and order of the values they represent must be consistent with those of the selected instance method.

The *nargs* – 1 words of arguments and *objectref* are popped from the operand stack. A new stack frame is created for the method being invoked, and *objectref* and the arguments are made the values of its first *nargs* words of local variables, with *objectref* in local variable 0, *arg1* in local variable 1, and so on. The new stack frame is then made current, and the Java Card Virtual Machine pc is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

### Runtime Exception

If *objectref* is null, the *invokespecial* instruction throws a `NullPointerException`.

## invokestatic

Invoke a class (static) method

### Format

<i>invokestatic</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

### Forms

*invokestatic* = 141 (0x8d)

### Stack

..., [*arg1*, [*arg2* ...]] ⇒  
...

### Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package, where the value of the index is  $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$ . The constant pool item at that index must be of type `CONSTANT_StaticMethodref`, a reference to a static method. The method must not be `<init>`, an instance initialization method, or `<clinit>`, a class or interface initialization method. It must be `static`, and therefore cannot be `abstract`. Finally, if the method is `protected`, then it must be either a member of the current class or a member of a superclass of the current class.

The resolved method includes the code for the method, an unsigned byte *nargs* that may be zero, and the method's modifier information.

The operand stack must contain *nargs* words of arguments, where the number of words of arguments and the type and order of the values they represent must be consistent with those of the resolved method.

The *nargs* words of arguments are popped from the operand stack. A new stack frame is created for the method being invoked, and the words of arguments are made the values of its first *nargs* words of local variables, with *arg1* in local variable 0, *arg2* in local variable 1, and so on. The new stack frame is then made current, and the Java Card Virtual Machine pc is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

## invokevirtual

Invoke instance method; dispatch based on class

### Format

<i>invokevirtual</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

### Forms

*invokevirtual* = 139 (0x8b)

### Stack

..., *objectref*, [*arg1*, [*arg2* ...]] ⇒

...

### Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package, where the value of the index is  $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$ . The constant pool item at that index must be of type `CONSTANT_VirtualMethodref`, a reference to a class and a virtual method token. The specified method is resolved. The method must not be `<init>`, an instance initialization method, or `<clinit>`, a class or interface initialization method. If the method is protected, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The resolved method reference includes an unsigned *index* into the method table of the resolved class and an unsigned byte *nargs* that must not be zero.

The *objectref* must be of type reference. The *index* is an unsigned byte which is used as an index into the method table of the class of the type of *objectref*. If the *objectref* is an array type, then the method table of class `Object` is used. The table entry at that index includes a direct reference to the method's code and modifier information.

The *objectref* must be followed on the operand stack by *nargs* – 1 words of arguments, where the number of words of arguments and the type and order of the values they represent must be consistent with those of the selected instance method.

The *nargs* – 1 words of arguments and *objectref* are popped from the operand stack. A new stack frame is created for the method being invoked, and *objectref* and the arguments are made the values of its first *nargs* words of local variables, with *objectref* in local variable 0, *arg1* in local variable 1, and so on. The new stack frame is then made current, and the Java Card Virtual Machine pc is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

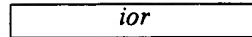
### Runtime Exception

If *objectref* is null, the *invokevirtual* instruction throws a `NullPointerException`.

## **ior**

Boolean OR `int`

### **Format**



### **Forms**

*ior* = 86 (0x56)

### **Stack**

..., *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2* ⇒  
..., *result.word1*, *result.word2*

### **Description**

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by taking the bitwise inclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

### **Notes**

If a virtual machine does not support the `int` data type, the *ior* instruction will not be available.

## **irem**

Remainder `int`

### **Format**

<i>irem</i>
-------------

### **Forms**

*irem* = 74 (0x4a)

### **Stack**

..., *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2*  $\Rightarrow$   
..., *result.word1*, *result.word2*

### **Description**

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is the value of the Java expression  $value1 - (value1 / value2) * value2$ . The *result* is pushed onto the operand stack.

The result of the *irem* instruction is such that  $(a/b) * b + (a \% b)$  is equal to *a*. This identity holds even in the special case that the dividend is the negative `int` of largest possible magnitude for its type and the divisor is  $-1$  (the remainder is 0). It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative and can be positive only if the dividend is positive. Moreover, the magnitude of the result is always less than the magnitude of the divisor.

### **Runtime Exception**

If the value of the divisor for a short remainder operator is 0, *irem* throws an `ArithmeticException`.

### **Notes**

If a virtual machine does not support the `int` data type, the *irem* instruction will not be available.

## **ireturn**

Return `int` from method

### **Format**

<i>ireturn</i>
----------------

### **Forms**

*ireturn* = 121 (0x79)

### **Stack**

..., *value.word1*, *value.word2* ⇒  
[empty]

### **Description**

The *value* must be of type `int`. It is popped from the operand stack of the current frame and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The virtual machine then reinstates the frame of the invoker and returns control to the invoker.

### **Notes**

If a virtual machine does not support the `int` data type, the *ireturn* instruction will not be available.

## **ishl**

Shift left *int*

### **Format**

<i>ishl</i>
-------------

### **Forms**

*ishl* = 78 (0x4e)

### **Stack**

..., *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2* ⇒  
..., *result.word1*, *result.word2*

### **Description**

Both *value1* and *value2* must be of type *int*. The values are popped from the operand stack. An *int result* is calculated by shifting *value1* left by *s* bit positions, where *s* is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

### **Notes**

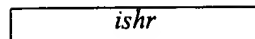
This is equivalent (even if overflow occurs) to multiplication by 2 to the power *s*. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

If a virtual machine does not support the *int* data type, the *ishl* instruction will not be available.

## ishr

Arithmetic shift right int

### Format



### Forms

*ishr* = 80 (0x50)

### Stack

..., *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2*  $\Rightarrow$   
..., *result.word1*, *result.word2*

### Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by shifting *value1* right by *s* bit positions, with sign extension, where *s* is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

### Notes

The resulting value is  $\lfloor (value1) / 2^s \rfloor$ , where *s* is *value2* & 0x1f. For nonnegative *value1*, this is equivalent (even if overflow occurs) to truncating `int` division by 2 to the power *s*. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

### Notes

If a virtual machine does not support the `int` data type, the *ishr* instruction will not be available.

## istore

Store int into local variable

### Format

<i>istore</i>
<i>index</i>

### Forms

*istore* = 42 (0x2a)

### Stack

..., *value.word1*, *value.word2* ⇒  
...

### Description

The *index* is an unsigned byte. Both *index* and *index* + 1 must be a valid index into the local variables of the current frame. The *value* on top of the operand stack must be of type `int`. It is popped from the operand stack, and the local variables at *index* and *index* + 1 are set to *value*.

### Notes

If a virtual machine does not support the `int` data type, the *istore* instruction will not be available.

## **istore\_<n>**

Store int into local variable

### **Format**

<i>istore_&lt;n&gt;</i>
-------------------------

### **Forms**

*istore\_0* = 51 (0x33)

*istore\_1* = 52 (0x34)

*istore\_2* = 53 (0x35)

*istore\_3* = 54 (0x36)

### **Stack**

..., *value.word1*, *value.word2* ⇒

...

### **Description**

Both <*n*> and <*n*> + 1 must be a valid indices into the local variables of the current frame. The *value* on top of the operand stack must be of type `int`. It is popped from the operand stack, and the local variables at *index* and *index* + 1 are set to *value*.

### **Notes**

If a virtual machine does not support the `int` data type, the *istore\_<n>* instruction will not be available.

## **i sub**

Subtract int

### **Format**

<i>isub</i>
-------------

### **Forms**

*isub* = 68 (0x44)

### **Stack**

..., *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2*  $\Rightarrow$   
..., *result.word1*, *result.word2*

### **Description**

Both *value1* and *value2* must be of type int. The values are popped from the operand stack. The int *result* is *value1* - *value2*. The *result* is pushed onto the operand stack.

For int subtraction,  $a - b$  produces the same result as  $a + (-b)$ . For int values, subtraction from zeros is the same as negation.

Despite the fact that overflow or underflow may occur, in which case the *result* may have a different sign than the true mathematical result, execution of an *isub* instruction never throws a runtime exception.

### **Notes**

If a virtual machine does not support the int data type, the *isub* instruction will not be available.

## itableswitch

Access jump table by int index and jump

### Format

<i>itableswitch</i>
<i>defaultbyte1</i>
<i>defaultbyte2</i>
<i>lowbyte1</i>
<i>lowbyte2</i>
<i>lowbyte3</i>
<i>lowbyte4</i>
<i>highbyte1</i>
<i>highbyte2</i>
<i>highbyte3</i>
<i>highbyte4</i>
<i>jump offsets...</i>

### Offset Format

<i>offsetbyte1</i>
<i>offsetbyte2</i>

### Forms

*itableswitch* = 116 (0x74)

### Stack

..., *index* ⇒  
...

### Description

An *itableswitch* instruction is a variable-length instruction. Immediately after the *itableswitch* opcode follow a signed 16-bit value *default*, a signed 32-bit value *low*, a signed 32-bit value *high*, and then  $high - low + 1$  further signed 16-bit offsets. The value *low* must be less than or equal to *high*. The  $high - low + 1$  signed 16-bit offsets are treated as a 0-based jump table. Each of the signed 16-bit values is constructed from two unsigned bytes as  $(byte1 \ll 8) | byte2$ . Each of the signed 32-bit values is constructed from four unsigned bytes as  $(byte1 \ll 24) | (byte2 \ll 16) | (byte3 \ll 8) | byte4$ .

The *index* must be of type `int` and is popped from the stack. If *index* is less than *low* or *index* is greater than *high*, then a target address is calculated by adding *default* to the address of the opcode of this *itableswitch* instruction. Otherwise, the offset at position  $index - low$  of the jump table is extracted. The target address is calculated by adding that offset to the address of the opcode of this *itableswitch* instruction. Execution then continues at the target address.

The target addresses which can be calculated from each jump table offset, as well as the one calculated from *default*, must be the address of an opcode of an instruction within the method that contains this *itableswitch* instruction.

## Notes

If a virtual machine does not support the `int` data type, the *itableswitch* instruction will not be available.

## iushr

Logical shift right int

### Format

<i>iushr</i>
--------------

### Forms

*iushr* = 82 (0x52)

### Stack

..., *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2*  $\Rightarrow$   
..., *result.word1*, *result.word2*

### Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by shifting the result right by *s* bit positions, with zero extension, where *s* is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

### Notes

If *value1* is positive and *s* is *value2* & 0x1f, the result is the same as that of *value1* >> *s*; if *value1* is negative, the result is equal to the value of the expression (*value1* >> *s*) + (2 << ~*s*). The addition of the (2 << ~*s*) term cancels out the propagated sign bit. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

If a virtual machine does not support the `int` data type, the *iushr* instruction will not be available.

## **ixor**

Boolean XOR *int*

### **Format**

<i>ixor</i>
-------------

### **Forms**

*ixor* = 88 (0x58)

### **Stack**

..., *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2*  $\Rightarrow$   
..., *result.word1*, *result.word2*

### **Description**

Both *value1* and *value2* must be of type *int*. The values are popped from the operand stack. An *int result* is calculated by taking the bitwise exclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

### **Notes**

If a virtual machine does not support the *int* data type, the *ixor* instruction will not be available.

## **jsr**

Jump subroutine

### **Format**

<i>jsr</i>
<i>branchbyte1</i>
<i>branchbyte2</i>

### **Forms**

*jsr* = 113 (0x71)

### **Stack**

...  $\Rightarrow$   
..., *address*

### **Description**

The *address* of the opcode of the instruction immediately following this *jsr* instruction is pushed onto the operand stack as a value of type `returnAddress`. The unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is  $(branchbyte1 \ll 8) \mid branchbyte2$ . Execution proceeds at that offset from the address of this *jsr* instruction. The target address must be that of an opcode of an instruction within the method that contains this *jsr* instruction.

### **Notes**

The *jsr* instruction is used with the *ret* instruction in the implementation of the `finally` clause of the Java language. Note that *jsr* pushes the address onto the stack and *ret* gets it out of a local variable. This asymmetry is intentional.

## **new**

Create new object

### **Format**

<i>new</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

### **Forms**

*new* = 143 (0x8f)

### **Stack**

...  $\Rightarrow$   
..., *objectref*

### **Description**

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package, where the value of the index is  $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$ . The item at that index in the constant pool must be of type `CONSTANT_Classref`, a reference to a class or interface type. The reference is resolved and must result in a class type (it must not result in an interface type). Memory for a new instance of that class is allocated from the heap, and the instance variables of the new object are initialized to their default initial values. The *objectref*, a reference to the instance, is pushed onto the operand stack.

### **Notes**

The *new* instruction does not completely create a new instance; instance creation is not completed until an instance initialization method has been invoked on the uninitialized instance.

## **newarray**

Create new array

### **Format**

<i>newarray</i>
<i>atype</i>

### **Forms**

*newarray* = 144 (0x90)

### **Stack**

..., *count* ⇒  
..., *arrayref*

### **Description**

The *count* must be of type `short`. It is popped off the operand stack. The *count* represents the number of elements in the array to be created.

The unsigned byte *atype* is a code that indicates the type of array to create. It must take one of the following values:

Array Type	<i>atype</i>
T_BOOLEAN	10
T_BYTE	11
T_SHORT	12
T_INT	13

A new array whose components are of type *atype*, of length *count*, is allocated from the heap. A reference *arrayref* to this new array object is pushed onto the operand stack. All of the elements of the new array are initialized to the default initial value for its type.

### **Runtime Exception**

If *count* is less than zero, the *newarray* instruction throws a `NegativeArraySizeException`.

### **Notes**

If a virtual machine does not support the `int` data type, the value of *atype* may not be 13 (array type = `T_INT`).

## **nop**

Do nothing

### **Format**

<i>nop</i>
------------

### **Forms**

*nop* = 0 (0x0)

### **Stack**

No change

### **Description**

Do nothing.

## pop

Pop top operand stack word

### Format

<i>pop</i>
------------

### Forms

*pop* = 59 (0x3b)

### Stack

..., *word* ⇒  
...

### Description

The top word is popped from the operand stack.

### Notes

The *pop* instruction operates on an untyped word, ignoring the type of data it contains.

## pop2

Pop top two operand stack words

### Format

<i>pop2</i>
-------------

### Forms

*pop2* = 60 (0x3c)

### Stack

..., *word2*, *word1* ⇒  
...

### Description

The top two words are popped from the operand stack.

The *pop2* instruction must not be used unless each of *word1* and *word2* is a word that contains a 16-bit data type or both together are the two words of a single 32-bit datum.

### Notes

Except for restrictions preserving the integrity of 32-bit data types, the *pop2* instruction operates on an untyped word, ignoring the type of data it contains.

## putfield\_<t>

Set field in object

### Format

<i>putfield_&lt;t&gt;</i>
<i>index</i>

### Forms

*putfield\_a* = 135 (0x87)  
*putfield\_b* = 136 (0x88)  
*putfield\_s* = 137 (0x89)  
*putfield\_i* = 138 (0x8a)

### Stack

..., *objectref*, *value* ⇒

...

OR

..., *objectref*, *value.word1*, *value.word2* ⇒

...

### Description

The unsigned *index* is used as an index into the constant pool of the current package. The constant pool item at the index must be of type `CONSTANT_InstanceFieldref`, a reference to a class and a field token. If the field is protected, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type reference
- *b* field must be of type byte or type boolean
- *s* field must be of type short
- *i* field must be of type int

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The *objectref*, which must be of type reference, and the *value* are popped from the operand stack. If the field is of type byte or type boolean, the *value* is truncated to a byte. The field at the offset from the start of the object referenced by *objectref* is set to the *value*.

### Runtime Exception

If *objectref* is null, the *putfield\_<t>* instruction throws a `NullPointerException`.

### Notes

If a virtual machine does not support the `int` data type, the *putfield\_i* instruction will not be available.

## putfield\_<t>\_this

Set field in current object

### Format

<i>putfield_&lt;t&gt;_this</i>
<i>index</i>

### Forms

*putfield\_a\_this* = 181 (0xb5)  
*putfield\_b\_this* = 182 (0xb6)  
*putfield\_s\_this* = 183 (0xb7)  
*putfield\_i\_this* = 184 (0xb8)

### Stack

..., *value* ⇒

...

OR

..., *value.word1*, *value.word2* ⇒

...

### Description

The currently executing method must be an instance method which was invoked using the *invokevirtual*, *invokeinterface* or *invokespecial* instruction. The local variable at index 0 must contain a reference *objectref* to the currently executing method's *this* parameter. The unsigned *index* is used as an index into the constant pool of the current package. The constant pool item at the index must be of type `CONSTANT_InstanceFieldref`, a reference to a class and a field token. If the field is protected, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type reference
- *b* field must be of type byte or type boolean
- *s* field must be of type short
- *i* field must be of type int

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The *value* is popped from the operand stack. If the field is of type byte or type boolean, the *value* is truncated to a byte. The field at the offset from the start of the object referenced by *objectref* is set to the *value*.

### Runtime Exception

If *objectref* is null, the *putfield\_<t>\_this* instruction throws a `NullPointerException`.

### Notes

If a virtual machine does not support the `int` data type, the *putfield\_i\_this* instruction will

not be available.

## **putfield\_<t>\_w**

Set field in object (wide index)

### **Format**

<i>putfield&lt;t&gt;_w</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

### **Forms**

*putfield\_a\_w* = 177 (0xb1)  
*putfield\_b\_w* = 178 (0xb2)  
*putfield\_s\_w* = 179 (0xb3)  
*putfield\_i\_w* = 180 (0xb4)

### **Stack**

..., *objectref*, *value* ⇒

...

OR

..., *objectref*, *value.word1*, *value.word2* ⇒

...

### **Description**

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package, where the value of the index is  $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$ . The constant pool item at the index must be of type `CONSTANT_InstanceFieldref`, a reference to a class and a field token. If the field is protected, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type reference
- *b* field must be of type byte or type boolean
- *s* field must be of type short
- *i* field must be of type int

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The *objectref*, which must be of type reference, and the *value* are popped from the operand stack. If the field is of type byte or type boolean, the *value* is truncated to a byte. The field at the offset from the start of the object referenced by *objectref* is set to the *value*.

### **Runtime Exception**

If *objectref* is null, the *putfield\_<t>\_w* instruction throws a `NullPointerException`.

### **Notes**

If a virtual machine does not support the `int` data type, the *putfield\_i\_w* instruction will not

be available.

## putstatic\_<t>

Set static field in class

### Format

<i>putstatic_&lt;t&gt;</i>
<i>indexbyte1</i>
<i>indexbyte2</i>

### Forms

*putstatic\_a* = 127 (0x7f)  
*putstatic\_b* = 128 (0x80)  
*putstatic\_s* = 129 (0x81)  
*putstatic\_i* = 130 (0x82)

### Stack

..., *value* ⇒

...

OR

..., *value.word1*, *value.word2* ⇒

...

### Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package, where the value of the index is  $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$ . The constant pool item at the index must be of type `CONSTANT_StaticFieldref`, a reference to a static field. If the field is protected, then it must be either a member of the current class or a member of a superclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type reference
- *b* field must be of type byte or type boolean
- *s* field must be of type short
- *i* field must be of type int

The width of a class field is determined by the field type specified in the instruction. The item is resolved, determining the class field. The *value* is popped from the operand stack. If the field is of type byte or type boolean, the *value* is truncated to a byte. The field is set to the *value*.

### Notes

If a virtual machine does not support the `int` data type, the *putstatic* instruction will not be available.

## ret

Return from subroutine

### Format

<i>ret</i>
<i>index</i>

### Forms

*ret* = 114 (0x72)

### Stack

No change

### Description

The *index* is an unsigned byte that must be a valid index into the local variables of the current frame. The local variable at *index* must contain a value of type `returnAddress`. The contents of the local variable are written into the Java Card Virtual Machine's pc register, and execution continues there.

### Notes

The *ret* instruction is used with the *jsr* instruction in the implementation of the `finally` keyword of the Java language. Note that *jsr* pushes the address onto the stack and *ret* gets it out of a local variable. This asymmetry is intentional.

The *ret* instruction should not be confused with the *return* instruction. A *return* instruction returns control from a Java method to its invoker, without passing any value back to the invoker.

## **return**

Return void from method

### **Format**

<i>return</i>
---------------

### **Forms**

*return* = 122 (0x7a)

### **Stack**

... ⇒  
[empty]

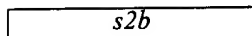
### **Description**

Any values on the operand stack of the current method are discarded. The virtual machine then reinstates the frame of the invoker and returns control to the invoker.

## s2b

Convert short to byte

### Format



### Forms

*s2b* = 91 (0x5b)

### Stack

..., *value*  $\Rightarrow$   
..., *result*

### Description

The *value* on top of the operand stack must be of type `short`. It is popped from the top of the operand stack, truncated to a `byte` *result*, then sign-extended to a `short` *result*. The *result* is pushed onto the operand stack.

### Notes

The *s2b* instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign as *value*.

## **s2i**

Convert short to int

### **Format**

<i>s2i</i>
------------

### **Forms**

*s2i* = 92 (0x5c)

### **Stack**

..., *value*  $\Rightarrow$   
..., *result.word1*, *result.word2*

### **Description**

The *value* on top of the operand stack must be of type `short`. It is popped from the operand stack and sign-extended to an `int` *result*. The *result* is pushed onto the operand stack.

### **Notes**

The *s2i* instruction performs a widening primitive conversion. Because all values of type `short` are exactly representable by type `int`, the conversion is exact.

If a virtual machine does not support the `int` data type, the *s2i* instruction will not be available.

## sadd

Add short

### Format

<i>sadd</i>
-------------

### Forms

*sadd* = 65 (0x41)

### Stack

..., *value1*, *value2*  $\Rightarrow$   
..., *result*

### Description

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. The `short result` is *value1* + *value2*. The *result* is pushed onto the operand stack.

If a *sadd* instruction overflows, then the result is the low-order bits of the true mathematical result in a sufficiently wide two's-complement format. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

## **saload**

Load short from array

### **Format**

<i>saload</i>
---------------

### **Forms**

*saload* = 38 (0x46)

### **Stack**

..., *arrayref*, *index*  $\Rightarrow$   
..., *value*

### **Description**

The *arrayref* must be of type reference and must refer to an array whose components are of type short. The *index* must be of type short. Both *arrayref* and *index* are popped from the operand stack. The short *value* in the component of the array at *index* is retrieved and pushed onto the top of the operand stack.

### **Runtime Exceptions**

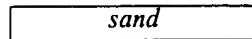
If *arrayref* is null, *saload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *saload* instruction throws an `ArrayIndexOutOfBoundsException`.

## **sand**

Boolean AND short

### **Format**



### **Forms**

*sand* = 83 (0x53)

### **Stack**

..., *value1*, *value2*  $\Rightarrow$   
..., *result*

### **Description**

Both *value1* and *value2* are popped from the operand stack. A short *result* is calculated by taking the bitwise AND (conjunction) of *value1* and *value2*. The *result* is pushed onto the operand stack.

## sastore

Store into short array

### Format

<i>sastore</i>
----------------

### Forms

*sastore* = 57 (0x39)

### Stack

..., *arrayref*, *index*, *value* ⇒

...

### Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `short`. The *index* and *value* must both be of type `short`. The *arrayref*, *index* and *value* are popped from the operand stack. The short *value* is stored as the component of the array indexed by *index*.

### Runtime Exception

If *arrayref* is null, *sastore* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *sastore* instruction throws an `ArrayIndexOutOfBoundsException`.

## **sconst\_<s>**

Push short constant

### **Format**

<i>sconst_&lt;s&gt;</i>
-------------------------

### **Forms**

*sconst\_m1* = 2 (0x2)

*sconst\_0* = 3 (0x3)

*sconst\_1* = 4 (0x4)

*sconst\_2* = 5 (0x5)

*sconst\_3* = 6 (0x6)

*sconst\_4* = 7 (0x7)

*sconst\_5* = 8 (0x8)

### **Stack**

... ⇒

..., <s>

### **Description**

Push the short constant <s> (-1, 0, 1, 2, 3, 4, or 5) onto the operand stack.

## **sdiv**

Divide short

### **Format**

<i>sdiv</i>
-------------

### **Forms**

*sdiv* = 71 (0x47)

### **Stack**

..., *value1*, *value2*  $\Rightarrow$   
..., *result*

### **Description**

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. The short *result* is the value of the Java expression *value1* / *value2*. The *result* is pushed onto the operand stack.

A short division rounds towards 0; that is, the quotient produced for short values in  $n/d$  is a short value  $q$  whose magnitude is as large as possible while satisfying  $|d \cdot q| = |n|$ . Moreover,  $q$  is a positive when  $|n| = |d|$  and  $n$  and  $d$  have the same sign, but  $q$  is negative when  $|n| = |d|$  and  $n$  and  $d$  have opposite signs.

There is one special case that does not satisfy this rule: if the dividend is the negative integer of the largest possible magnitude for the `short` type, and the divisor is  $-1$ , then overflow occurs, and the result is equal to the dividend. Despite the overflow, no exception is thrown in this case.

### **Runtime Exception**

If the value of the divisor in a short division is 0, *sdiv* throws an `ArithmeticException`.

## **sinc**

Increment local short variable by constant

### **Format**

<i>sinc</i>
<i>index</i>
<i>const</i>

### **Forms**

*sinc* = 89 (0x59)

### **Stack**

No change

### **Description**

The *index* is an unsigned byte that must be a valid index into the local variable of the current frame. The *const* is an immediate signed byte. The local variable at *index* must contain a short. The value *const* is first sign-extended to a short, then the local variable at *index* is incremented by that amount.

## **sinc\_w**

Increment local short variable by constant

### **Format**

<i>sinc_w</i>
<i>index</i>
<i>byte1</i>
<i>byte2</i>

### **Forms**

*sinc\_w* = 150 (0x96)

### **Stack**

No change

### **Description**

The *index* is an unsigned byte that must be a valid index into the local variable of the current frame. The immediate unsigned *byte1* and *byte2* values are assembled into a short *const* where the value of *const* is  $(byte1 \ll 8) \mid byte2$ . The local variable at *index*, which must contain a short, is incremented by *const*.

## **sipush**

Push short

### **Format**

<i>sipush</i>
<i>byte1</i>
<i>byte2</i>

### **Forms**

*sipush* = 19 (0x13)

### **Stack**

...  $\Rightarrow$   
..., *value1.word1*, *value1.word2*

### **Description**

The immediate unsigned *byte1* and *byte2* values are assembled into a signed `short` where the value of the short is  $(\text{byte1} \ll 8) \mid \text{byte2}$ . The intermediate value is then sign-extended to an `int`, and the resulting *value* is pushed onto the operand stack.

### **Notes**

If a virtual machine does not support the `int` data type, the *sipush* instruction will not be available.

## **sload**

Load short from local variable

### **Format**

<i>sload</i>
<i>index</i>

### **Forms**

*sload* = 22 (0x16)

### **Stack**

... ⇒  
..., *value*

### **Description**

The *index* is an unsigned byte that must be a valid index into the local variables of the current frame. The local variable at *index* must contain a short. The *value* in the local variable at *index* is pushed onto the operand stack.

## **sload\_<n>**

Load short from local variable

### **Format**

<i>sload_&lt;n&gt;</i>
------------------------

### **Forms**

*sload\_0* = 28 (0x1c)  
*sload\_1* = 29 (0x1d)  
*sload\_2* = 30 (0x1e)  
*sload\_3* = 31 (0x1f)

### **Stack**

...  $\Rightarrow$   
..., *value*

### **Description**

The <*n*> must be a valid index into the local variables of the current frame. The local variable at <*n*> must contain a short. The *value* in the local variable at <*n*> is pushed onto the operand stack.

### **Notes**

Each of the *sload\_<n>* instructions is the same as *sload* with an *index* of <*n*>, except that the operand <*n*> is implicit.

## slookupswitch

Access jump table by key match and jump

### Format

<i>slookupswitch</i>
<i>defaultbyte1</i>
<i>defaultbyte2</i>
<i>npairs1</i>
<i>npairs2</i>
<i>match-offset pairs...</i>

### Pair Format

<i>matchbyte1</i>
<i>matchbyte2</i>
<i>offsetbyte1</i>
<i>offsetbyte2</i>

### Forms

*slookupswitch* = 117 (0x75)

### Stack

..., *key* ⇒  
...

### Description

A *slookupswitch* instruction is a variable-length instruction. Immediately after the *slookupswitch* opcode follow a signed 16-bit value *default*, an unsigned 16-bit value *npairs*, and then *npairs* pairs. Each pair consists of a short *match* and a signed 16-bit *offset*. Each of the signed 16-bit values is constructed from two unsigned bytes as  $(byte1 \ll 8) | byte2$ .

The table *match-offset* pairs of the *slookupswitch* instruction must be sorted in increasing numerical order by *match*.

The *key* must be of type *short* and is popped from the operand stack and compared against the *match* values. If it is equal to one of them, then a target address is calculated by adding the corresponding *offset* to the address of the opcode of this *slookupswitch* instruction. If the *key* does not match any of the *match* values, the target address is calculated by adding *default* to the address of the opcode of this *slookupswitch* instruction. Execution then continues at the target address.

The target address that can be calculated from the offset of each *match-offset* pair, as well as the one calculated from *default*, must be the address of an opcode of an instruction within the method that contains this *slookupswitch* instruction.

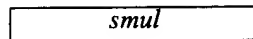
### Notes

The *match-offset* pairs are sorted to support lookup routines that are quicker than linear search.

## **smul**

Multiply short

### **Format**



### **Forms**

*smul* = 69 (0x45)

### **Stack**

..., *value1*, *value2* ⇒  
..., *result*

### **Description**

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. The short *result* is *value1* \* *value2*. The *result* is pushed onto the operand stack.

If a *smul* instruction overflows, then the result is the low-order bits of the mathematical product as a `short`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical product of the two values.

## **sneg**

Negate short

### **Format**

<i>sneg</i>
-------------

### **Forms**

*sneg* = 72 (0x4b)

### **Stack**

..., *value*  $\Rightarrow$

..., *result*

### **Description**

The *value* must be of type `short`. It is popped from the operand stack. The `short` *result* is the arithmetic negation of *value*,  $-value$ . The *result* is pushed onto the operand stack.

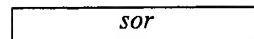
For short values, negation is the same as subtraction from zero. Because the Java Card Virtual Machine uses two's-complement representation for integers and the range of two's-complement values is not symmetric, the negation of the maximum negative `short` results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown.

For all short values  $x$ ,  $-x$  equals  $(\sim x) + 1$ .

## **SOR**

Boolean OR short

### **Format**



### **Forms**

*sor* = 85 (0x55)

### **Stack**

..., *value1*, *value2* ⇒  
..., *result*

### **Description**

Both *value1* and *value2* must be of type short. The values are popped from the operand stack. A short *result* is calculated by taking the bitwise inclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

## **srem**

Remainder short

### **Format**

<i>srem</i>
-------------

### **Forms**

*srem* = 73 (0x49)

### **Stack**

..., *value1*, *value2*  $\Rightarrow$   
..., *result*

### **Description**

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. The `short` *result* is the value of the Java expression  $value1 - (value1 / value2) * value2$ . The *result* is pushed onto the operand stack.

The result of the *irem* instruction is such that  $(a/b) * b + (a \% b)$  is equal to *a*. This identity holds even in the special case that the dividend is the negative short of largest possible magnitude for its type and the divisor is  $-1$  (the remainder is 0). It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative and can be positive only if the dividend is positive. Moreover, the magnitude of the result is always less than the magnitude of the divisor.

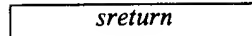
### **Runtime Exception**

If the value of the divisor for a short remainder operator is 0, *srem* throws an `ArithmeticException`.

## sreturn

Return short from method

### Format



### Forms

*sreturn* = 120 (0x78)

### Stack

..., *value* ⇒  
[empty]

### Description

The *value* must be of type short. It is popped from the operand stack of the current frame and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The virtual machine then reinstates the frame of the invoker and returns control to the invoker.

## **sshl**

Shift left short

### **Format**

<i>sshl</i>
-------------

### **Forms**

*sshl* = 77 (0x4d)

### **Stack**

..., *value1*, *value2* ⇒  
..., *result*

### **Description**

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. A `short` *result* is calculated by shifting *value1* left by *s* bit positions, where *s* is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

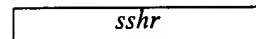
### **Notes**

This is equivalent (even if overflow occurs) to multiplication by 2 to the power *s*. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

## **sshr**

Arithmetic shift right short

### **Format**



### **Forms**

*sshr* = 79 (0x4f)

### **Stack**

..., *value1*, *value2*  $\Rightarrow$   
..., *result*

### **Description**

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. A `short` *result* is calculated by shifting *value1* right by *s* bit positions, with sign extension, where *s* is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

### **Notes**

The resulting value is  $\lfloor (value1) / 2^s \rfloor$ , where *s* is *value2* & 0x1f. For nonnegative *value1*, this is equivalent (even if overflow occurs) to truncating short division by 2 to the power *s*. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

## **sspush**

Push short

### **Format**

<i>sspush</i>
<i>byte1</i>
<i>byte2</i>

### **Forms**

*sspush* = 17 (0x11)

### **Stack**

...  $\Rightarrow$   
..., *value*

### **Description**

The immediate unsigned *byte1* and *byte2* values are assembled into a signed short where the value of the short is  $(byte1 \ll 8) | byte2$ . The resulting *value* is pushed onto the operand stack.

## **sstore**

Store short into local variable

### **Format**

<i>sstore</i>
<i>index</i>

### **Forms**

*sstore* = 41 (0x29)

### **Stack**

..., *value* ⇒  
...

### **Description**

The *index* is an unsigned byte that must be a valid index into the local variables of the current frame. The *value* on top of the operand stack must be of type `short`. It is popped from the operand stack, and the value of the local variable at *index* is set to *value*.

## **sstore\_<n>**

Store short into local variable

### **Format**

<i>sstore_&lt;n&gt;</i>
-------------------------

### **Forms**

*sstore\_0* = 47 (0x2f)  
*sstore\_1* = 48 (0x30)  
*sstore\_2* = 49 (0x31)  
*sstore\_3* = 50 (0x32)

### **Stack**

..., *value* ⇒  
...

### **Description**

The <*n*> must be a valid index into the local variables of the current frame. The *value* on top of the operand stack must be of type *short*. It is popped from the operand stack, and the value of the local variable at <*n*> is set to *value*.

## **ssub**

Subtract short

### **Format**

<i>ssub</i>
-------------

### **Forms**

*ssub* = 67 (0x43)

### **Stack**

..., *value1*, *value2*  $\Rightarrow$   
..., *result*

### **Description**

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. The `short` *result* is *value1* - *value2*. The *result* is pushed onto the operand stack.

For short subtraction,  $a - b$  produces the same result as  $a + (-b)$ . For short values, subtraction from zeros is the same as negation.

Despite the fact that overflow or underflow may occur, in which case the *result* may have a different sign than the true mathematical result, execution of a *ssub* instruction never throws a runtime exception.

## stableswitch

Access jump table by short index and jump

### Format

<i>stableswitch</i>
<i>defaultbyte1</i>
<i>defaultbyte2</i>
<i>lowbyte1</i>
<i>lowbyte2</i>
<i>highbyte1</i>
<i>highbyte2</i>
<i>jump offsets...</i>

### Offset Format

<i>offsetbyte1</i>
<i>offsetbyte2</i>

### Forms

*stableswitch* = 115 (0x73)

### Stack

..., *index*  $\Rightarrow$

...

### Description

A *stableswitch* instruction is a variable-length instruction. Immediately after the *stableswitch* opcode follow a signed 16-bit value *default*, a signed 16-bit value *low*, a signed 16-bit value *high*, and then  $high - low + 1$  further signed 16-bit offsets. The value *low* must be less than or equal to *high*. The  $high - low + 1$  signed 16-bit offsets are treated as a 0-based jump table. Each of the signed 16-bit values is constructed from two unsigned bytes as  $(byte1 \ll 8) | byte2$ .

The *index* must be of type `short` and is popped from the stack. If *index* is less than *low* or *index* is greater than *high*, then a target address is calculated by adding *default* to the address of the opcode of this *stableswitch* instruction. Otherwise, the offset at position *index* - *low* of the jump table is extracted. The target address is calculated by adding that offset to the address of the opcode of this *stableswitch* instruction. Execution then continues at the target address.

The target addresses which can be calculated from each jump table offset, as well as the one calculated from *default*, must be the address of an opcode of an instruction within the method that contains this *stableswitch* instruction.

## sushr

Logical shift right short

### Format

<i>sushr</i>
--------------

### Forms

*sushr* = 81 (0x51)

### Stack

..., *value1*, *value2*  $\Rightarrow$   
..., *result*

### Description

Both *value1* and *value2* must be of type short. The values are popped from the operand stack. A short *result* is calculated by sign-extending *value1* to 32 bits and shifting the result right by *s* bit positions, with zero extension, where *s* is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

### Notes

If *value1* is positive and *s* is *value2* & 0x1f, the result is the same as that of *value1* >> *s*; if *value1* is negative, the result is equal to the value of the expression (*value1* >> *s*) + (2 << ~*s*). The addition of the (2 << ~*s*) term cancels out the propagated sign bit. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

## swap\_x

Swap top two operand stack words

### Format

<i>swap_x</i>
<i>mn</i>

### Forms

*swap\_x* = 64 (0x40)

### Stack

..., *wordM+N*, ..., *wordM+1*, *wordM*, ..., *word1*  $\Rightarrow$   
..., *wordM*, ..., *word1*, *wordM+N*, ..., *wordM+1*

### Description

The unsigned byte *mn* is used to construct two parameter values. The high nybble, (*mn* & 0xf0) >> 4, is used as the value *m*. The low nybble, (*mn* & 0xf), is used as the value *n*. Permissible values for both *m* and *n* are 1-4.

The top *m* words on the operand stack are swapped with the *n* words immediately below.

The *swap\_x* instruction must not be used unless the ranges of words 1 through *m* and words *m+1* through *n* each contain either a 16-bit data type, two 16-bit data types, a 32-bit data type, a 16-bit data type and a 32-bit data type (in either order), or two 32-bit data types.

### Notes

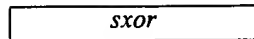
Except for restrictions preserving the integrity of 32-bit data types, the *swap\_x* instruction operates on untyped words, ignoring the types of data they contain.

If a virtual machine does not support the `int` data type, the permissible values for both *m* and *n* are 1 or 2.

## **sxor**

Boolean XOR short

### **Format**



### **Forms**

*sxor* = 87 (0x57)

### **Stack**

..., *value1*, *value2*  $\Rightarrow$   
..., *result*

### **Description**

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. A short *result* is calculated by taking the bitwise exclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

**This Page Blank (uspto)**

## Tables of Instructions

TABLE 8-1 Instructions by Opcode Value

0	nop	47	sstore_0	94	i2s	141	invokestatic
1	aconst_null	48	sstore_1	95	icmp	142	invokeinterface
2	sconst_m1	49	sstore_2	96	ifeq	143	new
3	sconst_0	50	sstore_3	97	ifne	144	newarray
4	sconst_1	51	istore_0	98	iflt	145	anewarray
5	sconst_2	52	istore_1	99	ifge	146	arraylength
6	sconst_3	53	istore_2	100	ifgt	147	athrow
7	sconst_4	54	istore_3	101	ifle	148	checkcast
8	sconst_5	55	aastore	102	ifnull	149	instanceof
9	iconst_m1	56	bastore	103	ifnonnull	150	sinc_w
10	iconst_0	57	sastore	104	if_acmpeq	151	inc_w
11	iconst_1	58	iastore	105	if_acmpne	152	ifeq_w
12	iconst_2	59	pop	106	if_scmpne	153	ifne_w
13	iconst_3	60	pop2	107	if_scmpne	154	iflt_w
14	iconst_4	61	dup	108	if_scmplt	155	ifge_w
15	iconst_5	62	dup2	109	if_scmpge	156	ifgt_w
16	bspush	63	dup_x	110	if_scmpgt	157	ifle_w
17	sspush	64	swap_x	111	if_scmple	158	ifnull_w
18	bipush	65	sadd	112	goto	159	ifnonnull_w
19	sipush	66	iadd	113	jsr	160	if_acmpeq_w
20	iipush	67	ssub	114	ret	161	if_acmpne_w
21	aload	68	isub	115	tableswitch	162	if_scmpeq_w
22	sload	69	smul	116	itableswitch	163	if_scmpne_w
23	iload	70	imul	117	slookupswitch	164	if_scmplt_w
24	aload_0	71	sdiv	118	lookupswitch	165	if_scmpge_w
25	aload_1	72	idiv	119	areturn	166	if_scmpgt_w
26	aload_2	73	srem	120	sreturn	167	if_scmple_w
27	aload_3	74	irem	121	ireturn	168	goto_w
28	sload_0	75	sneg	122	return	169	getfield_a_w
29	sload_1	76	ineg	123	getstatic_a	170	getfield_b_w
30	sload_2	77	sshl	124	getstatic_b	171	getfield_s_w
31	sload_3	78	ishl	125	getstatic_s	172	getfield_i_w
32	iload_0	79	sshr	126	getstatic_i	173	getfield_a_this
33	iload_1	80	ishr	127	putstatic_a	174	getfield_b_this
34	iload_2	81	sushr	128	putstatic_b	175	getfield_s_this
35	iload_3	82	iushr	129	putstatic_s	176	getfield_i_this
36	aload	83	sand	130	putstatic_i	177	putfield_a_w
37	baload	84	iand	131	getfield_a	178	putfield_b_w
38	saload	85	sor	132	getfield_b	179	putfield_s_w
39	iaload	86	ior	133	getfield_s	180	putfield_i_w
40	astore	87	sxor	134	getfield_i	181	putfield_a_this
41	sstore	88	ixor	135	putfield_a	182	putfield_b_this
42	istore	89	sinc	136	putfield_b	183	putfield_s_this
43	astore_0	90	iinc	137	putfield_s	184	putfield_i_this
44	astore_1	91	s2b	138	putfield_i	...	...
45	astore_2	92	s2i	139	invokevirtual	254	impdep1
46	astore_3	93	i2b	140	invokespecial	255	impdep2

TABLE 8-2 Instructions by Opcode Mnemonic

aaload	36	iaload	84	iload_0	32	putstatic_s	129
aastore	55	iastore	58	iload_1	33	ret	114
aconst_null	1	icmp	95	iload_2	34	return	122
aload	21	iconst_0	10	iload_3	35	s2b	91
aload_0	24	iconst_1	11	ilookupswitch	118	s2i	92
aload_1	25	iconst_2	12	imul	70	sadd	65
aload_2	26	iconst_3	13	ineg	76	saload	38
aload_3	27	iconst_4	14	instanceof	149	sand	83
anewarray	145	iconst_5	15	invokeinterface	142	sastore	57
areturn	119	iconst_m1	9	invokespecial	140	sconst_0	3
arraylength	146	idiv	72	invokestatic	141	sconst_1	4
astore	40	if_acmpeq	104	invokevirtual	139	sconst_2	5
astore_0	43	if_acmpeq_w	160	ior	86	sconst_3	6
astore_1	44	if_acmpne	105	irem	74	sconst_4	7
astore_2	45	if_acmpne_w	161	ireturn	121	sconst_5	8
astore_3	46	if_scmpgeq	106	ishl	78	sconst_m1	2
athrow	147	if_scmpgeq_w	162	ishr	80	sdiv	71
baload	37	if_scmpge	109	istore	42	sinc	89
bastore	56	if_scmpge_w	165	istore_0	51	sinc_w	150
bipush	18	if_scmpgt	110	istore_1	52	sipush	19
bspush	16	if_scmpgt_w	166	istore_2	53	sload	22
checkcast	148	if_scmpgt_w	166	istore_3	54	sload_0	28
dup	61	if_scmpne	111	isub	68	sload_1	29
dup_x	63	if_scmpne_w	167	itableswitch	116	sload_2	30
dup2	62	if_scmlt	108	iushr	82	sload_3	31
dup2_x	62	if_scmlt_w	164	ixor	88	slookupswitch	117
getfield_a	131	if_scmlt_w	164	jsr	88	smul	69
getfield_a_this	173	if_scmlt_w	164	jsr	88	sneg	75
getfield_a_w	169	if_scmlt_w	164	new	143	sor	85
getfield_b	132	ifeq	96	newarray	144	srem	73
getfield_b_this	174	ifeq_w	152	nop	0	sreturn	120
getfield_b_w	170	ifge	99	pop	59	sshl	77
getfield_i	134	ifge_w	155	pop2	60	sshr	79
getfield_i_this	176	ifgt	100	putfield_a	135	sspush	17
getfield_i_w	172	ifgt_w	156	putfield_a_this	181	sstore	41
getfield_s	133	ifgt_w	156	putfield_a_w	177	sstore_0	47
getfield_s_this	175	ifle	101	putfield_b	136	sstore_1	48
getfield_s_w	171	ifle_w	157	putfield_b_this	182	sstore_2	49
getstatic_a	123	ifle_w	157	putfield_b_w	178	sstore_3	50
getstatic_b	124	iflt	98	putfield_i	138	ssub	67
getstatic_i	126	iflt_w	154	putfield_i_this	184	stableswitch	115
getstatic_s	125	ifne	97	putfield_i_w	180	sushr	81
goto	112	ifne_w	153	putfield_s	137	swap_x	64
goto_w	168	ifnonnull	103	putfield_s_this	183	sxor	87
i2b	93	ifnonnull_w	159	putfield_s_w	179		
i2s	94	ifnull	102	putstatic_a	127		
iadd	66	ifnull_w	158	putstatic_b	128		
iaload	39	iinc	90	putstatic_i	130		
		iinc_w	151				
		iipush	20				
		iload	23				

# Glossary

---

**AID** is an acronym for Application Identifier as defined in ISO 7816-5.

**APDU** is an acronym for Application Protocol Data Unit as defined in ISO 7816-4.

**API** is an acronym for Application Programming Interface. The API defines calling conventions by which an application program accesses the operating system and other services.

**Applet** the basic unit of selection, context, functionality, and security in Java Card technology.

**Applet developer** refers to a person creating a Java Card applet using the Java Card technology specifications.

**Applet context.** The JCRE keeps track of the currently selected Java Card applet as well as the currently active Java Card applet. The currently active Java Card applet value is referred to as the Java Card applet context. When an instance method is invoked on an object, the Java Card applet execution context is changed to correspond to the Java Card applet that owns that object. When that method returns, the previous context is restored. Invocations of static methods have no effect on the Java Card applet execution context. The Java Card applet context and sharing status of an object together determine if access to an object is permissible.

**Atomic operation** is an operation that either completes in its entirety (if the operation succeeds) or no part of the operation completes at all (if the operation fails).

**Atomicity** refers to whether a particular operation is atomic or not and is necessary for proper data recovery in cases where power is lost or the card is unexpectedly removed from the CAD.

**ATR** is an acronym for Answer to Reset. An ATR is a string of bytes sent by the Java Card after a reset condition.

**CAD** is an acronym for Card Acceptance Device. The CAD is the device in which the card is inserted.

**Cast** is the explicit conversion from one data type to another.

**cJCK** is a test suite to verify the compliance of the implementation of the Java Card Technology specifications. The cJCK uses the JavaTest tool to run the test suite.

**Class** is the prototype for an object in an object-oriented language. A class may also be considered a set of objects which share a common structure and behavior. The structure of a class is determined by the class variables which represent the state of an object of that class and the behavior is given by a set of methods associated with the class.

Classes are related in a class hierarchy. One class may be a specialization (a “subclass”) of another (one of its “superclasses”), it may be composed of other classes, or it may use other classes in a client-server relationship.

**EEPROM** is an acronym for Electrically Erasable, Programmable Read Only Memory.

**EMV** is an acronym for Europay, MasterCard, and Visa. EMV is used to refer to the ICC specifications for payment systems.

**Framework** is the set of classes which implement the API. This includes core and extension packages. Responsibilities include dispatching of APDUs, applet selection, managing atomicity, and installing applets.

**Garbage collection** is the process by which dynamically allocated storage is automatically reclaimed during the execution of a program.

**GUI** is an acronym for Graphical User Interface. The GUI provides application control through the use of graphic images.

**ICC** is an acronym for Integrated Circuit Card.

**IDE** acronym for Interactive Development Environment. An IDE is a system for supporting the process of writing software which may include a syntax-directed editor, graphical tools for program entry, and integrated support for compiling the program and relating compilation errors back to the source.

**Instance variables**, also known as fields, represent a portion of an object’s internal state. Each object has its own set of instance variables. Objects of the same class will have the same instance variables, but each object can have different values.

**Instantiation**, in object-oriented programming, means to produce a particular object from its class template. This involves allocation of a data structure with the types specified by the template, and initialization of instance variables with either default values or those provided by the class’s constructor function.

**JAR** is an acronym for Java Archive. JAR is a platform-independent file format that combines many files into one.

**Java Card Runtime Environment (JCRE)** consists of the Java Card Virtual Machine, the framework, and the associated native methods.

**JC20RI** is an acronym for the Java Card 2.0 Reference Implementation.

**JCRE implementer** refers to a person creating a vendor-specific framework using the Java Card 2.0 API.

**JCVM** is an acronym for the Java Card Virtual Machine. The JCVM is the foundation of the OP card architecture. The JCVM executes byte code and manages classes and objects. It enforces separation between applications (firewalls) and enables secure data sharing.

**JDK** is an acronym for Java Development Kit. The JDK is a Sun Microsystems, Inc. product which provides the environment required for programming in Java. The JDK is available for a variety of platforms, but most notably Sun Solaris and Microsoft Windows®.

**MAC** is an acronym for Message Authentication Code. MAC is an encryption of data for security purposes.

**Method** is the name given to a procedure or routine, associated with one or more classes, in object-oriented languages.

**Namespace** is a set of names in which all names are unique.

**Object-Oriented** is a programming methodology based on the concept of an "object" which is a data structure encapsulated with a set of routines, called "methods," which operate on the data.

**Objects**, in object-oriented programming, are unique instances of a data structure defined according to the template provided by its class. Each object has its own values for the variables belonging to its class and can respond to the messages (methods) defined by its class.

**Package** is a namespace within the Java programming language and can have classes and interfaces. A package is the smallest unit within the Java programming language.

**Persistent object.** Persistent objects and their values persist from one CAD session to the next, indefinitely. Objects are persistent by default. Persistent object values are updated atomically using transactions. The term persistent does not mean there is an object-oriented database on the card or that objects are serialized/deserialized, just that the objects are not lost when the card loses power.

**PSE** is an acronym for Payment System Environment as described by the EMV specification.

**System configuration** refers to the combination of operating system platform and Java programming language tools.

**TCL** is an acronym for Tool Command Language. For more information on TCL, access the following URL: <http://2.brunel.ac.uk:8080/~csstddm/TCL2/TCL2.html>.

**Transaction** is an atomic operation where the developer defines the extent of the operation by indicating in the program code the beginning and end of the transaction.

**Transient object.** The values of transient objects do not persist from one CAD session to the next, and are reset to a default state at specified intervals. Updates to the values of transient objects are not atomic and are not effected by transactions.